Northumbria Research Link

Citation: Maatuk, Abdelsalam (2009) Migrating relational databases into object-based and XML databases. Doctoral thesis, Northumbria University.

This version was downloaded from Northumbria Research Link: http://nrl.northumbria.ac.uk/id/eprint/3374/

Northumbria University has developed Northumbria Research Link (NRL) to enable users to access the University's research output. Copyright © and moral rights for items on NRL are retained by the individual author(s) and/or other copyright owners. Single copies of full items can be reproduced, displayed or performed, and given to third parties in any format or medium for personal research or study, educational, or not-for-profit purposes without prior permission or charge, provided the authors, title and full bibliographic details are given, as well as a hyperlink and/or URL to the original metadata page. The content must not be changed in any way. Full items must not be sold commercially in any format or medium without formal permission of the copyright holder. The full policy is available online: http://nrl.northumbria.ac.uk/policies.html

Northumbria University NEWCASTLE

CUSTOMER SERVICE EXCELLENCE CSE

UniversityLibrary

MIGRATING RELATIONAL DATABASES INTO OBJECT-BASED AND XML DATABASES

By

Abdelsalam Amraga Maatuk

A thesis submitted in partial fulfilment
of the requirements of the
Northambria University at Newcastle
for the degree of
Doctor of Philosophy

Research undertaken in the School of Computing, Engineering and Information Sciences
Newcastle

July 2009

I would like to thank Almighty Allah, for granting me the patience and capabilities needed to complete this dissertation.

I would like then to thank my parents for their encouragement and prayers for me. Without them this work would never have come into existence. Finally I thank my wife and my children for their support and patience during this work, and this dissertation is therefore dedicated to them

Table of Contents

List of Tables

List of Figures

Abstract

Rapid changes in information technology, the emergence of object-based and WWW applications, and the interest of organisations in securing benefits from new technologies have made information systems re-engineering in general and database migration in particular an active research area. In order to improve the functionality and performance of existing systems, the re-engineering process requires identifying and understanding all of the components of such systems. An underlying database is one of the most important component of information systems. A considerable body of data is stored in relational databases (RDBs), yet they have limitations to support complex structures and user-defined data types provided by relatively recent databases such as object-based and XML databases. Instead of throwing away the large amount of data stored in RDBs, it is more appropriate to enrich and convert such data to be used by new systems. Most researchers into the migration of RDBs into object-based/XML databases have concentrated on schema translation, accessing and publishing RDB data using newer technology, while few have paid attention to the conversion of data, and the preservation of data semantics, e.g., inheritance and integrity constraints. In addition, existing work does not appear to provide a solution for more than one target database. Thus, research on the migration of RDBs is not fully developed. We propose a solution that offers automatic migration of an RDB as a source into the recent database technologies as targets based on available standards such as ODMG 3.0, SQL4 and XML Schema. A canonical data model (CDM) is proposed to bridge the semantic gap between an RDB and the target databases. The CDM preserves and enhances the metadata of existing RDBs to fit in with the essential characteristics of the target databases. The adoption of standards is essential for increased portability,

flexibility and constraints preservation.

This thesis contributes a solution for migrating RDBs into object-based and XML databases. The solution takes an existing RDB as input, enriches its metadata representation with the required explicit semantics, and constructs an enhanced relational schema representation (RSR). Based on the RSR, a CDM is generated which is enriched with the RDB's constraints and data semantics that may not have been explicitly expressed in the RDB metadata. The CDM so obtained facilitates both schema translation and data conversion. We design sets of rules for translating the CDM into each of the three target schemas, and provide algorithms for converting RDB data into the target formats based on the CDM.

A prototype of the solution has been implemented, which generates the three target databases. Experimental study has been conducted to evaluate the prototype. The experimental results show that the target schemas resulting from the prototype and those generated by existing manual mapping techniques were comparable. We have also shown that the source and target databases were equivalent, and demonstrated that the solution, conceptually and practically, is feasible, efficient and correct.

Declaration

I hereby declare that the work contained in this dissertation has not been submitted for any other degree and that, to the best of my knowledge and belief, it is all my sole work except where indicated.

Name: Abdelsalam Maatuk

Signature:

Date: 1 July 2009

Acknowledgements

I would like to take this opportunity to thank all those who have contributed to the completion of this work.

I would like to express my heartfelt gratitude to my supervisors, Dr M. Akhtar Ali and Dr. Nick Rossiter, for their invaluable guidance and assistance. Dr. Akhtar helped me to identify all crucial elements of my research work from the beginning stages to the final draft. Much of the work would have been impossible without his generosity, suggestions and support. Special thanks go to him for his contribution to providing a challenging yet comfortable and friendly working environment. Dr. Rossiter has also been very supportive throughout my research, and I am especially thankful to him for the time he devoted to the critical reviewing of my work and for his pertinent academic advice.

These acknowledgements would not be complete without thanking my family, including my parents, wife, children, brothers and sisters, for their love and support. I cannot thank my father and mother enough for their continuous encouragement and prayers. I would like to thank my wife for her understanding and encouragement, without which it would have been much harder to accomplish this work.

I would also like to sincerely acknowledge the financial support of the General Peoples Committee for Higher Education, Libya, which gave me the opportunity to study in the UK, and to the Cultural Affairs Department, Libyan People's Bureau, London, for their help and support.

Many thanks go to all staff members of the School of Computing, Engineering and Information Sciences for their support with all study resources.

Abdelsalam Maatuk
1 July 2009

Chapter 1

Introduction

Over the last three decades, information technology has attracted a significant investment due to its fast evolution, importance and extensive acceptance. Rapid changes in information technology have created challenges for organisations to respond quickly to the requirements of the global market in order to organise work more flexibly and efficiently. There has been a demand to keep pace with these changes, take advantage of the benefits of new technologies and meet the needs of customers. As a natural result of these changes and requirements, information systems re-engineering has become an important research and practical issue.

Systems re-engineering involves a wide range of tasks associated with understanding, transforming and redesigning existing information systems, while at the same time utilising as much of existing systems as possible. System re-engineering can be defined as a process of discovering how an existing information system works [Tari and Stokes, 1997; Alhajj, 2003]. This demands identifying the semantics of all the components of the existing system and the relationships among them, and converting them into design-level components such as entities, attributes and relationships [Comyn-Wattiau and Akoka, 1996]. These design-level components can be used to create new applications or redesign an existing system to meet new requirements. One of the most important components of an information system is the underlying database [Alhajj, 2003]. However, databases cannot be easily replaced. In other words, it might not be possible to re-write database applications every time the user wants to switch to or respond to a new technology. Consequently, reverse engineering techniques and tools started to emerge in order to resolve some problems in the context of databases, such as existing database maintenance, integration and conversion. The main reason

behind these initiatives is to avoid throwing away a huge amount of structured and verified data present in existing systems.

Database applications re-engineering is not a trivial task because its aim is to understand and recover the domain semantics of an existing database. Forward engineering and reverse engineering are considered to be the two directions of the re-engineering process [Lee and Yoo, 2000]. However, reverse engineering is of increasing importance in the context of using existing systems together with novel database systems or of migrating from traditional to more recent systems [Fahrner and Vossen, 1995a; Müller et al., 2000; Hohenstein, 2000; Lee and Yoo, 2000].

Database reverse engineering (DBRE) aims to recover the conceptual schema, which expresses the domain semantics of a database. Domain semantics may be explicitly and implicitly expressed in the schema and application programs [Chiang et al., 1993, 1996]. However, a database can be re-engineered independently of any application programs [Hüsemann, 1998; Henrard et al., 2002]. Database forward engineering (DBFE) aims to obtain the target physical schema from the conceptual schema obtained from DBRE [Fahrner and Vossen, 1995a]. However, database re-engineering is much complicated than schema translation. The most important part of databases, the typed data, are to be converted and utilised within the new environment.

Database systems have been designed in accordance with specific requirements such as structuring and performance. Database applications are implemented with database management systems (DBMSs), which are chosen to fulfil these requirements. Most traditional database applications are based on traditional DBMSs, i.e., relational database management systems (RDBMSs) as they have been quite successful in handling simple but large amounts of data. Codd [1970] formally proposed the relational data model, which was further developed through implementation by IBM and other companies. In the 1980s, RDBMSs revolutionised data management and business information systems. In the decades since, relational databases (RDBs) have been applied in a number of areas and accepted as a solution for storing and retrieving data due to their maturity. Large bodies of data are stored in RBDs nowadays.

The increasing popularity of new object-based and World Wide Web (WWW) technologies and non-traditional applications (e.g., multimedia, geographical information systems, computer aided design, etc.) can be considered to be among the most significant recent changes in information technology. These novel technologies have been dominant in the area of information systems due to their productivity, flexibility

and extensibility. Object-oriented databases (OODBs) [Kim, 1991], object-relational databases (ORDBs) [Stonebraker et al., 1999] and the extensible markup language (XML) [XML, 2008], which all support various diverse concepts, have been proposed in order to fulfil the demands of complex applications that require rich data types. Consequently, new breed of DBMSs have started to emerge in the market, providing more functionality and flexibility. However, as the majority of databases are still stored and maintained in RDBMSs, therefore, it is expected that the need to convert such RDBs into the technologies that have emerged recently will grow substantially [Lee et al., 2001; Fong et al., 2006].

The drawbacks of RDBMSs in supporting complex data structures, user-defined data types and new applications have led to the development of OODBs and object-oriented DBMSs (OODBMSs). Another reason for progress in OODBs is the promise of object-oriented (OO) technology in general such as OO analysis and design life cycle using unified modeling language (UML) [OMG, 2009] and OO programming languages (OOPLs). Intensive efforts have been made by the object database management group (ODMG) in proposing a standard known as ODMG 3.0 [Cattell and Barry, 2000]. This standard is a "jump start" in the object database industry, in which the core aspects of OODBs, e.g., the portability for persistent object storage specification can be precisely made [Cattell and Barry, 2000].

The need for additional data modelling feature has also been recognised by RDBMS vendors, leading to the appearance of object-based relational systems [Elmasri and Navathe, 2006]. An ORDB has potential because it has a relational technology base and appends object features. The main goal of its design was to incorporate both the robust transaction and performance management features of the relational model and the features of flexibility, scalability and support for rich data types of OO models. Some of these features are defined in the SQL3 standard [Eisenberg and Melton, 1999], e.g., user-defined types and inheritance, and others may be expressed in SQL4 [Pardede et al., 2003; Eisenberg et al., 2004], e.g., collection-valued attributes.

The emergence of new applications such as e-commence has led to the development of languages and tools for exchanging and publishing RDB data over the Web, and so Web databases have become a main stream research area. XML is nowadays used as a database at content level and as a dominant standard at hypertext level [Kappel et al., 2001]. Because of the increasing importance of XML, the limitations and problems associated with XML-enabled RDBs in handling XML documents, have led to the

emergence of native-XML databases to handle XML documents.

In view of this, many companies desire to adopt new technologies and re-engineer all (or part) of their existing information systems to benefit from these technologies. Most companies have been developing new applications in accordance with the new technology. However, large bodies of data are still stored in conventional RDBs. On the one hand, given the amount of money and time invested in existing RDB systems, companies do not want to throw away their data [Alhajj and Polat, 2001]. On the other hand, existing RDBs cannot be integrated seamlessly with the newly developed applications due to the semantic gap between different paradigms, which is known as the object-relational impedance mismatch [Behm et al., 1997; Ambler, 2003].

Generally speaking, there are three scenarios as to how newly developed applications could benefit from existing RDBs: (a) migrating RDBs to the relatively newer and richer (i.e., object-based/XML) databases, (b) developing new applications on top of existing RDBs, and (c) creating new databases using newer technologies (i.e., object-based/XML) and establishing connections with existing RDBs.

The first scenario, which is the focus of this dissertation, is to migrate RDBs into the richer databases. Database migration is concerned with the process of converting schema and data from a source RDB, as a one-time conversion, into a target database to be managed and handled in its new environment. The source schema is enriched semantically and translated into a target schema, and the data stored in the source database are converted into a target database based on the new schema.

The second alternative scenario to bridge the semantic gap between RDBs and the relatively newer technologies is that object-based/XML applications are developed on top of RDBs, where data can be processed in object/XML form and stored in relational form based on the concept of object for programs and RDB for persistence [Takahashi and Keller, 1994; Liu et al., 2003]. Application developers have to write large amounts of code to map objects in programs into tuples in an RDB, which might be time-consuming to write and execute. Another solution for this scenario is through *mapping middleware* that links non-relational concepts to data stored in RDBs.

The third alternative scenario is to keep the existing RDBs for data retrieval purposes only, and to develop a new database application according to the novel technology.

The goal is to establish a connection between RDBs and other databases thus allowing the applications built on top of the new DBMS to access both relational and object/XML DBMSs [Orenstein and Kamber, 1995]. This would give an impression that all data are stored in one database. This is achieved using a special type of software called *gateways*, which support connectivity between DBMSs. Most commercial DBMSs provide flexibility on gateways construction among heterogeneous databases. However, while gateways can provide interoperability between the existing system and other systems, they also give rise to the problem of maintaining data consistency between the two systems [Bisbal et al., 1999].

The difference between gateways and mapping middleware is that, in the former, objects are persistently stored in the new target database system, whereas in the latter, objects are created and handled in the normal way but are stored in an RDB. However, these two scenarios focus on schema translation rather than data conversion, and data stored in an RDB are retained. In addition, mapping using middleware requires schema mapping time, which may lead to slow performance. The advantage of migrating RDBs into the newer target databases is that once objects have been converted they can be stored and retrieved directly in the target databases without any need for translation layers, hence saving development time and increasing performance.

Database migration is very important in encouraging organisations to move to a new technology. As information is one of the most precious resources for organisations, database migration scenario have to be presented prior to any request to move to a new technology [Alhajj and Polat, 2001]. In addition, the features of standards such as ODMG 3.0 [Cattell and Barry, 2000], SQL4 [Pardede et al., 2003] and XML Schema [W3C, 2008] are supported by many DBMSs with powerful query languages, which make it necessary to try migrating existing data into the new environment. Therefore, several organisations are willing to go with this approach and give the new databases a try, however, the question is which of the new databases is most appropriate to adopt in migrating the existing database?

The above discussion highlights the reasons as to why migrating an RDB to OODB, ORDB and XML is needed. Existing work does not appear to provide a solution for more than one target database, and none of the previous proposals can be considered as a method for converting an RDB into an ORDB. The approach proposed in this dissertation is a database migration solution, comprising three basic phases. In

the first phase, the method produces an intermediate canonical data model (CDM), which is enriched with integrity constraints and data semantics of an existing RDB. The CDM so obtained is translated into OODB/ORDB/XML schemas in the second phase. Data conversion is the third phase, in which RDB data are converted into their equivalents in a new database environment. The solution is more beneficial compared to the existing approaches as it produces three different output databases based on the user choice. A system architecture has been designed and a prototype implemented to demonstrate the migration process and to facilitate a proof of its concepts. An experimental study has been conducted to evaluate the prototype by checking the results it provides regarding the correctness and completeness of the solution and its concepts. In this dissertation, the proposed solution and its prototype implementation is referred to as MIGROX (migrating RDB into object-based and XML databases).

The rest of this chapter is organised as follows. Section 1.1 provides a brief introduction to database migration. Section 1.2 summarises the motivation for the work described in this dissertation. Section 1.3 outlines the aims of the research and the contributions stemming from it. The organisation of the dissertation is outlined in Section 1.4.

1.1 Database Migration

Migration of database applications is a process in which all components of a source database application are converted into their equivalents in a target database environment. This involves translating the source schema into the target one, converting source data into the target database format, migrating application programs on top of the new and non-relational DBMS, and mapping queries and update operations into their equivalents in the target platform. However, the conversion of application programs and queries is a software engineering task and is, therefore, outside the scope of this research. Therefore, it is assumed that database migration includes schema translation and data conversion.

1.1.1 Schema Translation

A schema of an existing data model S_1 can be translated into an equivalent target schema S_2 expressed in the target data model through applying a set of mapping rules [Ramanathan and Hodges, 1997]. The generation of a well-designed target schema depends on the flexibility of these rules. Each rule maps a specific construct, e.g., attribute or relationship. Both schemas should hold equivalent semantics. The translation of a source schema to a target schema consists of two steps. The first step, i.e, DBRE, aims to recover the conceptual schema, e.g., an entity relationship (ER) model [Chen, 1976], which expresses the explicit and implicit data semantics of the source schema. Explicit semantics involve relations, attributes, keys and data dependencies. It is necessary to extract extra semantics that are not expressed explicitly in RDBs (e.g., inheritance relationship, cardinality constraints, relationship names). The process requires the analysis of database schemas, data instances and the source code of programs including queries and update operations that use stored data, in order to understand and extract their structure and meaning [Zhang and Fong, 2000]. In order to discover such data and constraints, developers may require specific tools and program understanding techniques in this domain to ease this process. The second step, i.e., DBFE, aims to obtain the target physical schema from the conceptual schema obtained in the first step. The first step is generally known as the semantic enrichment process, which is essential for database migration and database integration [Hohenstein and Körner, 1996; Hohenstein and Plesser, 1996]. However, the source schema can be translated directly to a target one without intermediate representation [Fahrner and Vossen, 1995b]. An expert user or a tool might be required to provide the missing semantics or to refine the results to exploit the concepts of the target database [Premerlani and Blaha, 1994; Fahrner and Vossen, 1995b].

1.1.2 Data Conversion

Data conversion is a process for converting data instances from the source database into the target database. Data stored as tuples in an RDB are converted into complex objects/literals in object-based databases or elements in XML documents. This involves extracting and restructuring RDB data, and then reloading the converted data into a target database in order to populate the schema generated earlier during

the schema translation process [Fong, 1997].

1.2 Motivation of the Research

Several factors have motivated the investigation described in this dissertation. Many organisations have stored their data in RDBs and aspire to take advantage of databases that have emerged more recently. Instead of discarding existing RDBs or building non-relational applications on top of them, it is generally preferable and beneficial to convert existing relational data into a new environment. However, the question is: which of the new databases is most appropriate to move to? So there is a need for a method that deals with database migration from RDB to OODB/ORDB/XML in order to provide an opportunity for exploration, experimentation and comparison among alternative database technologies. The method should assist in evaluating and choosing the most appropriate target database to adopt for non-relational applications to be developed according to the required functionality, performance and suitability. This could help further increase the acceptance of such newer and richer databases among enterprises and practitioners. However, the difficulty facing this method is that it targets multiple database models which are conceptually different. There is no existing canonical model that can be used as an intermediate stage for schema and data conversion from input RDB to various output targets.

Research so far has focused on diverse areas of RDB migration. The migration of an RDB into its equivalent has only been accomplished in the existing literature using two databases. The first is an RDB, as a source database, and another target database represents the result of the migration process. Most existing methods for migrating RDBs into OODBs focus on schema translation, whereas the work for migrating into XML focuses on generating a document type definition (DTD) [DTD, 2009] schema and data. Moreover, all research on the generation of ORDBs is focused on design rather than migration. The existing work does not appear to provide a solution for more than one target database for either schema or data conversion. In addition, none of the existing proposals can be considered as a method for migrating an RDB into an ORDB.

Numerous methods have been proposed for transforming logical data models into conceptual data models [Hainaut, 1991; Fonkam and Gray, 1992; Malki et al., 2002; Alhajj, 2003]. Well-known models such as an ER, extended ER (EER) and UML are

usually used in these methods as conceptual models. However, a large body of research has concentrated on database design, for example, the generation of ORDB/XML schemas from ER and UML diagrams [Kleiner and Lipeck, 2001; Laforest and Boumediene, 2003; Vela and Marcos, 2003]. Approaches focusing on schema translation are usually proposed when data stored in RDBMSs are required to be processed in object/XML forms [Orenstein and Kamber, 1995; Funderburk et al., 2002]. RDBs data can be accessed using object-to/from-relational and XML-to/from-relational mapping techniques, which link an RDB to non-relational applications or by using *gateways* that support connectivity between RDBMS and other DBMSs [Abu-Hamdeh et al., 1994]. Hence, queries and operations are converted into SQL and the results are translated into objects in non-relational databases [Orenstein and Kamber, 1995]. Large number of prototypes and tools exist to enable non-RDB applications to share data with object schema (e.g., Penguin [Takahashi and Keller, 1993, 1994]), and to publish RDB data as XML documents (e.g., SilkRoute [Fernandez et al., 2000], XPERANTO [Carey et al., 2000a] and XTABLES [Funderburk et al., 2002]). In addition, most commercial DBMSs provide flexibility in mapping and gateways construction among heterogeneous databases. However, such tools and many other currently available in the industry are based on the structure-mapping approach, which can only convert RDB relations into corresponding targets in a target database as a one-to-one mapping without preservation of data semantics and constraints [Fong et al., 2006].

Only limited work has attempted data conversion, and even these efforts have some drawbacks. Owing to the focus on schema rather than data, many proposals either ignore data conversion or assume working on virtual target databases (using object-relational mapping or gateways middleware), and data remain stored in RDBs. Using middleware may lead to slow performance, making the process expensive at run-time because of the dynamic mapping of tuples to complex objects [Behm et al., 2000]. Data, which are the most important part of databases need to be converted and utilised within new environments.

Numerous studies have only captured the structure of existing RDBs, whereas implicit data semantics are mostly ignored. Some types of semantics (e.g., inheritance, aggregation, integrity constraints) have often been ignored, mainly due to their lack of support in either source or target data models [Narasimhan et al., 1993; Fahrner and Vossen, 1995b]. For instance, the ER model and DTD lack support for inheritance.

Although inheritance relationships could be realised in an RDB, they have been either ignored or only briefly mentioned without analysis of their different types. Translating inheritance relationships from RDBs to object-based/XML databases needs more attention. As the latest standards (i.e., an ODMG 3.0, SQL4 and an XML Schema) have gained a wide acceptance in recent years for more independence, it is important that database migration methods generate target databases according to these standards. The adoption of standards is essential for increased portability, interoperability, flexibility and constraints preservation [Elmasri and Navathe, 2006].

In the majority of published work, a database is generated that is either flat like relational or has a deep level of clustering/nesting [Fong, 1997; Lee et al., 2001]. However, object-based model features and the hierarchical form of XML model are usually missed in such work. It would be desirable to avoid the flattened form and to reduce the levels of clustering for object structure to the lowest possible, in order to increase the utilisation that the target models provide and to avoid unnecessary redundancy. This requires the preservation of the semantics of the source RDB and their relocation into an intermediate conceptual representation, which takes into account the relatively richer data model of the target database environment.

An RDB migration process aims to migrate a source RDB into *an equivalent* target database. Database equivalence verifies the effectiveness and validity of the migration method. To demonstrate the effectiveness and validity of the migration method, a prototype has to be developed to realise the method's algorithms and translation rules. In addition, experimental studies should be designed to evaluate the prototype by checking the results it generates. The concepts, models and algorithms used in the migration method need to be checked in term of correctness. In addition, the completeness of the schema translation and data conversion rules also needs to be verified. Database equivalence should include data semantics, data instances and integrity constraints. However, experimental studies to verify that the source and target databases are equivalent are still not enough. Only a few approaches have highlighted this problem, concentrating on the validation of the generated schema [Lee et al., 2001; Fong and Cheung, 2005], whereas the evaluation of the equivalences between the source and target databases regarding data content and integrity constraints are often ignored.

The investigation described in this dissertation was motivated by the lack of a complete, fully implemented and empirically validated solution which deals with migration from RDBs to OODB, ORDB and XML covering levels of both schema and data.

This dissertation proposes such a complete solution, which has been implemented as a prototype. The prototype shows that the concepts used in the solution can be implemented in terms of programming languages. The prototype is then evaluated by checking the equivalence between the input and the output of the prototype, and by comparing its output with the results of existing methods.

1.3 Thesis Statement

1.3.1 Aims and Objectives

This research aims to devise a method for migrating an RDB into object-based and XML databases, thus solving the problems that were outlined in the previous section. We claim that an integrated method can be developed based on a CDM which enriches an RDB schema with the required implicit and explicit semantics and takes into account the characteristics of the target databases in order to automatically migrate an existing RDB into object-based and XML databases. The following objectives have been set forth to achieve these aims:

1. To review existing approaches to migrating RDBs into object-based and XML databases, considering their capabilities, weaknesses and limitations,
2. To devise a comprehensive solution to the problem of RDB migration, aiming to provide complete migration to an OODB, ORDB and XML,
3. To implement a prototype system based on (2), and
4. To evaluate the prototype by testing its results, looking at our achievements and reflecting on the results.

1.3.2 Main Contributions

The contributions of this research can be summarised as follows.

1. A review is provided of existing approaches and techniques related to database conversion problem. The capabilities and limitations of database conversion techniques, concentrating on migrating an RDB into object-based and XML databases, are analysed and some unresolved problems are determined.

A perspective based on this review has been published [Maatuk et al., 2008b].

2. A solution is offered to the problem of migrating RDBs into OODB, ORDB and XML in the form of MIGROX, which has the following features:

   - It offers a CDM as an intermediate stage for the better preservation of integrity constraints and data semantics. Using CDM, a new target database can be generated without referring to an existing RDB each time another target database needs to be generated. This provides reading and enriching an RDB once for multiple subsequent usages. The CDM definition and the algorithms developed to generate it from an RDB have been recently published [Maatuk et al., 2008c].
   - It translates the CDM into an object-based and XML schema according to standards and data definition languages (DDLs), leading to more portability and flexibility.
   - It provides a data conversion technique which automatically converts an existing RDB data into any of the target databases based on CDM. In other words, the CDM determines and manages the conversion of data into targets.
   - It is implemented as a prototype, which is experimentally evaluated to demonstrate the effectiveness and correctness of the solution.
   - An overview of this solution has been published [Maatuk et al., 2008a]. The complete migration process from RDBs into ORBDs has been accepted for publication [Maatuk et al., 2010].

1.4 Outline of the Dissertation

The remainder of the dissertation is structured as follows.

Chapter 2 provides the background information necessary to understand the work described in the rest of this dissertation. The areas covered by this dissertation and introduced in this chapter include relational, object-based and XML data models, and database systems, current standards and query languages.

Chapter 3 surveys the recent literature about various research trends relevant to RDB migration solutions. The chapter presents an analysis of approaches and techniques

used in this context, including construction of object views on top of RDBs, database integration and database migration. A categorisation is presented of selected work in the literature, involving translation techniques used for the problem of database conversion, concentrating on migrating an RDB as a source into object-based and XML databases as targets. Database migration from the source into each of the targets is discussed and critically evaluated, including the semantic enrichment, schema translation and data conversion.

An overview of the MIGROX solution is presented in Chapter 4. The chapter introduces the concepts and assumptions underlying this solution. The chapter provides an introduction to the three phases of MIGROX, i.e., semantic enrichment, schema translation and data conversion. This includes the formal notations, with which CDM and target data models are defined as well as some of the preliminary results of MIGROX. In addition, the chapter presents the algorithm used to infer the necessary RDB schema information that is then used for CDM generation. Chapters 5-7 constitute a precise description of the solution outlined in this chapter.

The focus of Chapter 5 is the CDM generation algorithm as the output of the semantic enrichment process, which is the first phase of the database migration. The chapter describes in detail how to identify CDM constructs using information provided by RDB metadata and how to generate the relationships and cardinalities among classes using data instances.

Chapter 6 focuses on the schema translation phase. Three sets of translation rules designed for translating CDM into its equivalent targets schemas are described in this chapter. Algorithms are developed for producing target schemas according to these rules.

Algorithms for converting RDB data into target databases are presented in Chapter 7. By resolving a number of problems with existing techniques, the extraction, transformation and loading of existing RDB data into target database formats are described in detail.

Chapter 8 explains how the prototype was developed, by implementing the algorithms and main concepts presented in Chapters 4-7. This chapter discusses the development environment and the reasons for choosing the software used in the implementation. System architecture and the main modules of the prototype are illustrated. The prototype modules and their components and how they work and communicate with

each other are described in detail.

Chapter 9 discusses how an experimental study was conducted to evaluate the prototype. The correctness of the CDM and schema translation algorithms are checked by comparing the target schemas resulting from the prototype and those generated by existing manual-based mapping techniques. Using query-based experiments, source and target data equivalences are checked by observing variations in results regarding data content and integrity constraints. A number of questions and challenges encountered are also outlined.

Chapter 10 concludes the dissertation by summarising the main contributions of the research and identifying possible directions for future work.

Chapter 2

Contemporary Databases

This chapter provides an illustrative background to the main concepts of relational, object-based and XML data models, in order to get a better understanding of this dissertation and the process RDB migration. The chapter is restricted to the basic concepts of the data models relevant to the database migration process. It includes introductions to the models, recent standards and DBMSs, the advantages and disadvantages of each model, and finally a comparison of their concepts. This investigation establishes the foundation of the proposed CDM, based on which the migration process is carried out. Further insight in such data models can be found in main database books [Kim, 1991; Rumbaugh et al., 1990; Gogolla, 1994; Stonebraker et al., 1999; Cattell and Barry, 2000; Date, 2002; Connolly and Begg, 2002; Graves and Goldfarb, 2002; Valentine et al., 2002; Elmasri and Navathe, 2006; Garcia-Molina et al., 2008].

A *database* is a structured collection of data describing the characteristics of people and things. Between the physical database and the users of the system is a layer known as a database server or a *database management system* (DBMS). The database is used by the application system and managed by a DBMS [Date, 2002]. A DBMS is a collection of programs that enables users to create databases, manipulate their data and translate them into information. There are many DBMSs, such as Oracle, DB2, and MS SQL server. Compared to filing systems, which support the storage of large amounts of data, database systems have the advantages of speed, accuracy and accessibility. A DBMS is a shell surrounding one or more databases throughout various interactions such as data maintenance, data retrieval, and data control. A *data model* is a set of conceptual tools to describe data, including data semantics, relationships and constraints. It aims to organise the stored data logically and physically within

the database design phase for efficient management. Many data models, such as relational, object oriented and XML data models, and others have played important roles since the first appearance of DBMSs. Each of these data models describes the data and relationships among them in its own way. Each model has advantages that may help to overcome the shortcomings of the other models.

This chapter is organised as follows. Section 2.1 provides an introduction to relational data model and the SQL, and Section 2.2 reviews the main concepts of object-oriented data models and their standards. Section 2.3 provides a background to an object-relational data model followed by an introduction to SQL3 and SQL4, while the XML data model and XML Schema language are introduced in Section 2.4. A comparison of the concepts underlying these data models is provided in Section 2.5, and Section 2.6 summarises the chapter and points to what follows.

2.1 Relational Data Model

The relational data model, introduced by Codd [1970], represents a database as a collection of *relations* (i.e., tables of values); hence the name relational database. Later, the ER model [Chen, 1976], which is currently used as the main conceptual model, was proposed for graphically structuring a relational model. Extensions to this model, i.e., EER [Gogolla, 1994] have been proposed in the '80s and '90s because of its widespread use in practice. Data are structured and stored in RDBs in two-dimensional tables. The relational model focuses on tuple-oriented information and primitive data types. Each table consists of a number of rows, called *tuples*, each of which consists of a collection of related values. Each column (or field) in a table is called an *attribute*. Each attribute has a data type (e.g., integer, char, date). Each attribute occupies one column and each tuple occupies one row. The primary unit of data is a data item (e.g., student_ID), which is said to be atomic. A set of data items of the same type is called a *domain*. For example, a table might contain student data with columns representing first name, surname, address and course_id and each row of the table represents information about a specific student. Each tuple is uniquely identifiable and independent of other tuples in the same table. Organising data into tables/relations is a logical view of how data are represented by a DBMS. How data is actually stored (i.e., the physical view) is not visible to the user and is independent of the logical view. Database designers define tables according to business requirements

to put related tables together to form a database [Cattell and Barry, 2000].

2.1.1 Constraints

There are some restrictions on the data stored in RDBs, which are called *constraints*. The constraints avoid inconsistencies among tuples in databases. The constraints in RDBs include implicit, explicit, semantic and data dependency constraints [Date, 2002; Connolly and Begg, 2002; Elmasri and Navathe, 2006]. Implicit constraints are inherent in the data model for the characteristics of relations, e.g., relationships among tuples. Explicit constraints can be expressed in the schema using the DDL, e.g., integrity constraints. Semantic constraints cannot be directly expressed in the schema, hence they are hidden in application programs or data content. Data dependencies test whether or not the RDB is designed perfectly using the normalisation process.

The integrity constraints are: key, entity integrity, referential integrity, domain, and null. These constraints should be enforced by RDBMSs at each instance of insertion, updating or deletion of data to or from the tables. Each row in a table represents one tuple, where the order and position of rows is insignificant. A tuple must be unique, where the uniqueness is achieved by having an identifier. An identifier can be a single attribute or a set of attributes. The identifier is called the primary key. Each table should have a primary key, the value of which cannot be null. It is the database designer's job to determine the best candidate keys from which to extract primary keys, typically numbers (e.g., student_ID). A tuple in a table may refer to another tuple in the same or other tables. The reference can be in the form of one or a set of attributes. The set of attributes used for referencing the primary key is called a foreign key. A foreign key is an attribute or a set of attributes of one table whose values match a primary key of another table. Inclusion dependency can be identified through referential integrity between the foreign key values in one table and the primary key values in one or more tables. Normalisation is a technique used in database design to reduce redundancy, data anomalies and poor data integrity.

2.1.2 Standardization of SQL

The structured query language (SQL) represents the standard for RDBs. SQL offers two sub-languages: a data definition language (DDL) and a data manipulation language (DML). The DDL defines schemas and explicit constraint specifications. The DML consists of certain statements, which are defined based on relational algebra for querying, inserting, updating, and deleting RDB data. Furthermore, the DML has many query types for manipulating data of the field values of tuples, from simple one-table queries to complex multiple-table queries, which include nesting, join and union. Today, there are many SQL products. The most important SQL standard is that which has been adopted by the international organisation for standardizations (ISO) and the American national standards institute (ANSI) [Connolly and Begg, 2002]. A revised SQL standard, before it is extended to support the object concepts, is called SQL2. Some example of the RDBMSs available are Oracle 7, DB2 and Microsoft Access.

2.1.3 Advantages and Disadvantages of RDBs

Although RDBs are sufficient for managing the storage of large capacity of primitive data types, and their SQL is relatively easy to learn, they are not strong enough to model real world problems and new application areas. RDBs have difficulty in representing complex structures, operations, images, and video stream collections [Kim, 1991]. Moreover, they cannot handle applications such as computer aided design, spatio-temporal databases, and other applications that involve complex data interrelationships [Devarakonda, 2001]. The relational data model does not allow for user-defined data types that can be defined based on primitive or other pre-defined data types. Relations may not represent entities in the real world, and the inheritance relationships are not directly supported. One object may be normalised into many relations, and consequently queries become complex and cumbersome to execute because the `select` and `join` operations need to be used frequently to reform the object. New operations cannot be added to the system in the relational model since it is limited to the generic SQL operations. The complex data structures handled by non-relational applications are mismatched with the data types in the RDB system, where concepts of richer data semantics such as inheritance and encapsulation need to be translated into lower relational semantics, e.g., the problem known as

object-relational impedance mismatch. RDBMS programmers may spend much time in coding the mapping of objects into RDBs for persistence [Leavitt, 2000]. All these limitations have led to the development of new extensions known as object-relational DBMSs (ORDBMS) [Stonebraker et al., 1999] such as IBM DB2 Extender and Oracle 11_g.

2.2 Object-Oriented Data Model

The demand to represent complex data structures has motivated the development of the object-oriented (OO) systems. The OO models offer concepts that enable a better modeling of real world problems to conceptual schemas [Kim, 1991; Rumbaugh et al., 1990]. OO modelling supports a tight integration of codes and data with flexible data types and hierarchical relationships between them, where each entity in an object model is called an object instance. With the growing need to represent, manipulate and store complex data, OODBMSs have many advantages over RDBMSs. OODB systems combine the features of OO technology with database capability. All OO concepts are supported, including classes (or abstract data types), data encapsulation, inheritance, polymorphism, operations and complex objects. In addition, OODBs provide persistence for class instances.

- **Classes:** A class groups the common features of a set of objects. It has two parts: an interface and implementation. The internal data structure is hidden and external operations can be applied to objects, leading to encapsulation.
- **Objects:** Real world objects are represented as persistent/entity objects with a number of levels of complexity and an inheritance hierarchy. An object has a state (value) and behaviour (operations). The state of an object can only be changed using its operations. Each object is an instance of one or more classes, and has a unique identifier (OID), which is used as a reference to the object. Related objects are connected using their OIDs. The OIDs are independent of data contained and are system-generated. In addition, they are neither visible to the user nor change even though their contents have changed.
- **Encapsulation:** Code and data are packaged together to form objects and protecting them from access by other code defined outside. Through its defined interface, an object can be manipulated, and thus its structure is hidden.

- **Inheritance:** A class can be derived from other classes. This is a powerful mechanism, which lets a class inherits the attributes and operations of a previously defined parent class, so that it can be extended with additional properties. Databases then have to manage object storage according to a class hierarchy.
- **Polymorphism:** Depending on the run-time use of objects, this concept permits the same operation name to be associated with different kinds of implementation.
- **Persistence:** Unlike with OOPLs where objects are transient in nature, objects in databases are required to be persistently stored so that they can be accessed later.

2.2.1 The ODMG Standard

As with any new technology, the principal drawback of OODBMSs when they first emerged was their lack of standardisation. OODBs are not based on a data model as accepted as the relational model. A number of standards have been proposed to encourage the development of OODBMSs, such as manifestos and ODMG. Manifestos have three releases, the latest of which adds some features of OO to databases based on SQL [Darwen and Date, 1995]. However, they have been developed by different groups belonging to different institutions. Intensive efforts have been made by the ODMG group who proposed a standard known as the ODMG-93; now revised into ODMG 3.0 released in 1999 [Cattell and Barry, 2000]. The standard is supported by most of the OODBMSs industry. The many products supporting this standard include: ObjectStore [ObjectStore, 2009], Objectivity [Objectivity, 2009], and Lambda-DB [Fegaras, 2008].

The ODMG 3.0 standard defines portability for persistent object storage specification, aiming to allow applications written with different OOPLs to access data stored in OODBMSs in a uniform way. Portability is the capability for accessing different OODBMSs (that support the same OO paradigm and standards) using one application program with insignificant modifications [Elmasri and Navathe, 2006]. In addition, utilising an ODMG standard, interoperability is achieved. That is, a particular application program can access data stored in diverse DBMSs, even though they are based on fundamentally different paradigms such as RDB and OO systems. An ODMG allows transparent integration to OOPLs like Java, and thus developers have

the ability to work within a native OO environment. Developers can handle and store objects directly without any mapping tools. However, ODMG 3.0 is not supported by a stand-alone query language, but depends on OOPLs. The standard has an object definition language (ODL) and an object query language (OQL), the syntax of which is similar to the SQL with some additional OO features such as identity, inheritance and polymorphism, and it supports constructed types like `Bag`, `Set` and `List` [Cattell and Barry, 2000]. The ODMG 3.0 standard consists of the following:

- Object model, which is proposed to allow applications portability among OODBMSs. It determines the data structure concepts, such as objects, attributes, relationships, and inheritance. It also defines data types, e.g., string, collections and `struct`.
- Object specification language, which consists of the ODL and object interchange format (OIF). The ODL is designed to support the semantic constructs of the ODMG object model. It is used to define the schema of an ODMG compliant database, independently from any programming language. An OIF is a specification language for object representation and for loading objects into/from files.
- It has a number of bindings to OOPLs, i.e., C++, Smalltalk and Java, which define several classes in order to access, create, retrieve and manipulate objects from applications.
- OQL, which is for retrieving data from object base. It is non-procedural in design, based on SQL to work with the ODMG binding programming languages for updating and querying objects. OQL supports the ODMG system, including the support of object identity, path expressions, inheritance and methods. OQL provides an interface for posting ad hoc queries.

The ODMG Object Model

An ODMG object model provides a standard for OODBs via the ODL and OQL [Cattell and Barry, 2000]. An ODL is used to specify an OODB schema that conform to the ODMG model. An OQL works with ODMG OOPLs as a query language extending SQL with object concepts.

Objects types and literal types: Objects and literals are the basic modelling primitives. Each object has a unique identity, a state, behaviour, and optionally a name. Each object must have a unique, system-generated and unchangeable OID; this is called a mutable object. A literal is an immutable object, which is a value that cannot change and does not have an OID (e.g., an integer value). A literal definition specifies only the state of objects, it does not specify their behaviour. An object encapsulates its state and behaviour, so that rich semantics and integrity are achieved. Objects are categorised into object types. A set of the instances of an object type is called an `extent` which holds all persistent objects of a class. The `extent` is a database entry point for updating and querying the objects of a corresponding class. A type specification may be either an interface or a class.

State and behaviour of objects: A user-defined type is specified in ODL as an `interface` or `class`. The interface defines only the abstract object behaviour, whereas the class defines both abstract state and behaviour. A class is instantiable, whereas an interface is not and its state cannot be inherited. Moreover, unlike with classes, objects cannot be created corresponding to interfaces. For both types, the state of the object is represented by its properties defined by attributes and relationships. A collection of operations represent the behaviour of an object. An attribute represents a fact about an object that has a descriptive name and value. A class definition may specify a unique `key`. One (or more) attributes (or relationships) can be selected as a `key`. The attribute may be simple or complex. The relationship represents links among objects. It has two descriptive names: one labels a relationship path and the other labels the inverse relationship path. A relationship is represented as a pair of inverse references, using the `relationship` and `inverse` keywords. However, a relationship can be defined as an attribute or using methods. Operations have parameters, return values and may raise error handling exceptions.

Built-in collection objects: An ODMG object model has three literal types: simple, structured and collection. Simple literals are basic types, e.g., an integer and string. Structured literals are built-in types that are structured using a tuple constructor such as `Date` and `Time`. Collections are type constructors used to define collections of other types, and to store multiple values in a single attribute. A collection literal is a value of a set of objects, but the collection has no identifier. A

structured object represents a structured entity as a single object. Structured objects are either collection types or structured types. Built-in collection types are `Set`, `Bag`, `List`, `Array` and `Directory`. `Set` and `Bag` are unordered, whereas `List` and `Array` are ordered collections. A structured type contains a fixed number of objects and is defined using the `struct` type generator.

Inheritance: The ODMG model defines two types of inheritance: the `IS-A` and `EXTENDS` relationships. An `IS-A` relationship defines behaviour inheritance between object types in which interfaces and classes can inherit other interfaces using the ":" symbol. In addition, multiple-inheritance is supported in the `IS-A` relationship. However, the `EXTENDS` relationship is a single inheritance relationship, and it defines the inheritance of state and behaviour among classes only.

2.2.2 Advantages and Disadvantages of OODBs

Unlike with RDB systems where complex data structures are flattened into tables, OODBs are suitable for applications that deal with complex relationships along with data objects. Besides, due to their ability to handle various types of multimedia and WWW applications, and supporting OO technology, industry observers have been widely attracted to OODBs. There will be fewer mismatch difficulties, a unification of database applications with seamless data models and programming language environments when an OOPL is implemented on top of OODBMSs. As a consequence, applications use more natural data structures, there is less code to develop, and development time and maintenance expense are reduced [Devarakonda, 2001]. Moreover, it has been predicted that OODBMSs will become the basic database technology, superseding RDBMSs. OODBs are of practical use in removing the redundancy of data and update anomaly problem, which might exist in RDBs. On the other hand, OODBMSs do have drawbacks. Query optimisation is very complex, there may be scalability problems, and they are not able to support large-scale systems [Devarakonda, 2001]. OODBMSs need skilled users to manage data, as they are not as readily understood as relational. In addition, they lack a mathematical foundation and a standard ad hoc query language.

2.3 Object-Relational Data Model

The object-relational technology is relatively recent, nevertheless, it is not a new technology on its own right [Connolly and Begg, 2002]. ORDBs have potential because they merge relational modelling and OO concepts. The main objective of their design was to incorporate both the robust transaction and performance management features of RDBs, and the OO model features of flexibility, scalability and support for rich data types [Stonebraker et al., 1999]. Together with the ability to handle alphanumerical data types, ORDBs can handle multimedia data types. Developers can work with tabular relational structures and DDL with the possibility of object management. Furthermore, the development of ORDBs was triggered by the growth of OOPLs to avoid the mismatch between these languages and DBMSs.

The object-relational specification extends a relational model to include OO capabilities. These include abstract data types (ADTs), structured and collection data types, identifiers and references, path expressions and inheritance, and operations. An ORDB consists of a set of tables, called *typed tables* as they can be created based on pre-defined ADTs. Each tuple (row object) of a table has a system-generated OID, through which relationships among objects are established. OIDs can be defined from primary keys and can be user-generated. The 1NF, the basic rule of RDB design, has been relaxed in the object-relational models, so that an attribute can contain a collection of data types. Structured and multi-valued literal types can be defined as attributes within ADTs; but they do not have OIDs as they are not object types.

2.3.1 The SQL3 Standard

Many vendors have created their SQL languages to be used for their own products to capture the OO concepts. This has led to a lack of standards that can be used by all ORDB users, and the differences amongst these products may be significant. As a consequence, the development of a standard called SQL-1999 (or SQL3) has been motivated, which became the foundation for most ORDBMSs [Pardede et al., 2003]. SQL3 [Eisenberg and Melton, 1999] and its successor SQL4 [Eisenberg et al., 2004; Pardede et al., 2003] are extensions of SQL-92 to include the new features of OO concepts. SQL3 supports ADTs, including OIDs, methods, inheritance, polymorphism, and the support of a binary large objects (BLOBs), character large objects (CLOBs) and collection types. Theses concepts are discussed in Section 2.2.

- **Abstract data types:** SQL3 allows the users to define ADTs according to their needs, including object specification, i.e., attributes and methods. An ADT allows values in tables to be associated with methods (encapsulation). Attributes are encapsulated within ADTs and accessed via methods. The attributes of an ADT can be defined as another ADT, representing a complex attribute. Besides, a table can be defined based on an ADT.

- **Reference type:** A row in a table is an object that is uniquely identified by a column called identity, containing an OID, according to which an object differs from other objects. The type of this column is a `ref`, which is used for relationship participation.

- **Structured type:** SQL3 adds a new structured type called a `row` that allows a number of attributes to be represented as a single column entry. It differs from ADTs as it has no OID and no methods associated with it.

- **Collection type:** SQL 3 supports collection types which represent multi-valued attributes as a single type. `Array` and `set` are among these types. An `Array` can hold multiple values to be referenced as a whole, however, its element must have similar types, have limited size and be ordered, whereas `set` is an unordered collection which does not allow duplicates. SQL4 adds `list` and `multiset` collection types that allow duplicates. The `list` is ordered, whereas `multiset` is unordered.

- **Inheritance:** Simple inheritance is supported in SQL3/SQL4 via the `under` keyword for both ADTs and tables. Sub-types and super-types can participate in an inheritance hierarchy. A sub-type inherits all attributes and functions of its top level super-types. In addition, a sub-type can define its own attributes and functions and override inherited functions. A sub-table inherits more from its super-table, including columns, rows, key, triggers and methods. Every row in a sub-table corresponds exactly to one row in the super-table and every row in the super-table corresponds to at most one row in a sub-table.

2.3.2 Advantages and Disadvantages of ORDBs

The main advantages of ORDBs are their huge scalability, the possibility of reuse and sharing, and their compatibility with existing RDBMSs. ORDBMSs are designed to

have large storage capacities to satisfy large organisations wanting to manage massive databases. ORDBMSs are expected to outsell RDBMSs, given their supplementary object capabilities. The four main features of an ORDBMS are: base data type extension, the support of complex objects, inheritance, and rule systems [Stonebraker et al., 1999]. Although ORDBMSs have many advanced features and most DBMS vendors expect them to become the market leader and resolve many of the known weaknesses of RDBMSs, they still have some disadvantages. ORDBMS architecture is not regarded as appropriate for high-speed web-applications, given its complexity and increased cost [Devarakonda, 2001]. Moreover, the simplicity of the relational model may be lost as SQL4 is already complex.

2.4 XML Data Model

Not all data are handled in structured databases [Elmasri and Navathe, 2006]. Data might be collected in databases where they do not have to be constrained by the schema. These types of data are called semi-structured data, which mix a schema with a data instance. Examples of the semi-structured data model are object exchange model (OEM) [OEM, 2009] and XML. XML has become the best choice for data exchange on the Web [Layman et al., 1998; Graves and Goldfarb, 2002; Valentine et al., 2002]. It is self-describing, according to which documents can be structured into complex levels of nesting and can be validated against a schema definition. XML is a powerful model because it extends simple user-defined tags to more levels with complex structures and relationships such as aggregation and inheritance. Because of the increasing importance of XML, native-XML databases have emerged to handle XML documents, such as Lore [Goldman et al., 2000; McHugh et al., 1997], XMLSpy [Altova XMLSpy, 2008] and eXist-db [eXist-db, 2009]. This section provides an overview of XML from the point of view of databases.

XML is a text-based markup language for structured documents, which is a subset of the standard generalised markup language (SGML) [XML, 2008]. It is designed to facilitate interoperability with SGML and HTML and is fast becoming the standard for the World Wide Web consortium (W3C) for data interchange over the Internet [W3C, 2009]. As with HTML, data in XML is identified as a rooted tree using tags that are known as “markup”. However, unlike HTML, XML tags describe the data, rather than specifying how to represent it.

An XML document contains several constructs such as namespaces, elements, attributes, tags and values. An element is a unit of XML data enclosed by tags which can enclose other elements. A tag is a piece of text that describes elements or data. Unlike data, tags are surrounded by angle brackets (< and >). An attribute is a qualifier on the tag that provides further information. Values occur as instances of elements/attributes.

XML is described as a mechanism for specifying the semantics of the data. Therefore, an XML document consists of a schema and data that can be combined in one document. However, schema specifications can be stored in a separate file. The schema describes the data structure and constraints using one of the XML schema languages. The selection of the language depends on its ability to fit the application requirements. Dozens of XML schema languages have been proposed, such as DTD and XML Schema by W3C, and XDR by Microsoft. A comparative analysis of some of these languages can be found elsewhere [Lee and Chu, 2000].

2.4.1 XML Schema Language

XML Schema language is a standard that provides a sophisticated means for describing the structures and constraints of XML schema and instance documents [W3C, 2008]. XML Schema borrows concepts from RDB models such as key and integrity constraints, and other concepts from OO models such as inheritance, references, data collections and user-defined data types. Moreover, it provides a wider range of built-in data types, where users can define their own simple/complex data types using `restriction` and `extension` keywords. The essential components of XML Schema are explained below.

Namespaces and annotations: XML Schema allows the concept of namespaces to avoid conflicts among elements and attributes naming. Elements and attributes are associated in the root element with namespaces, using the `xmlns` attribute and a prefix, e.g., `xs:`. Then, all types defined in the schema must be prefixed by `xs:`. Annotations provide useful comments and information in the schema document. For example, the `documentation` indicates the language used in the document, with the attribute `lang` (e.g., `lang` = '`en`'). The '`en`' stands for the English language.

Elements and attributes: An element structure with name, type, default value, possibly a set of identity-constraint definitions, occurrences and user annotations, can be specified using an element declaration [W3C, 2008]. Elements that are declared under the root of the schema are called global elements, which can be referenced using the '`ref`' attribute. Elements that are declared inside types or model groups are called local elements. In contrast to global elements that have to be unique in the schema, local elements can be declared with the same name and different types if they are not declared at the same level. Consequently, global elements should be declared for reusing, whereas local elements are preferable where elements are unlikely to be reused. Element structure should be defined as strictly hierarchical using the `sequence` composer if the order is significant or defined loosely using the `all` composer otherwise. The `name` attribute is mandatory for global elements, and the `type` attribute defines element types as simple or complex. Simple types are used to define attributes and elements that contain only data using the `simpleType` tag, whereas complex type definitions are used to define elements that contain child-elements or attributes using the `complexType` tag. The `minOccurs` and `maxOccurs` attributes specify the minimum and maximum occurrences of an element. Like elements, attributes can be declared globally or locally with name, type, occurrence, and default information. However, attributes cannot involve other elements as children because they must be defined as simple types, and the order in defining attributes is not significant.

Identity constraints: XML Schema provides comprehensive support to represent integrity constraints. Using the XPath expression [Berglund et al., 2007], it is possible to specify constraints that correspond to unique values, primary keys and foreign keys in RDBs. The tags `unique`, `key` and `keyref` are used to define unique, key and key reference, respectively. The XPath expression `selector` defines the scope of a constraint, and the `field` defines the elements or attributes that represent the constraint (e.g., a unique attribute). Key reference is defined by adding the `refer` attribute to specify the referenced primary key constraint name. In the case of composite keys, the `field` declaration is repeated for each element (column) in the key. Unique and key elements emphasise uniqueness concerning the content identified by `selector` of the tuples determined by `fields` XPath expression(s). The `keyref` concerns matching the tuples determined by `fields` in `selector` by those of the referenced key.

Attribute and model groups: A group of attributes for many elements can be defined as a collection called an attribute group that can be referenced. An attribute group is similar to a global attribute; however, it is a clustered set of attributes, whereas a global attribute is an individual attribute declaration. Similar to an attribute group, a model group can be defined for components reuse. However, it is a mechanism for creating a group of elements that must include `sequence`, `all`, or `choice` compositors. The advantage of model groups is to avoid type derivations; however, they differ from complex types as they cannot define attributes as children and they are limited to type derivations.

Inheritance: Elements and attributes are allowed in XML Schema to be extended or restricted using derivation and substitution mechanisms. Content models can be extended using `extension` and can be restricted using `restriction`. A simple type cannot be extended, but can be restricted using `restriction`, `list` or `union` tags. However, a complex type can be restricted, allowing only one or more new attributes to be defined using `restriction`, and can be extended using `extension` when extra attributes or element can be added.

2.4.2 XQuery

The XQuery query language has been developed by the W3C as a standard for XML [Boag et al., 2009]. The language differs from SQL as it organises queries by a so-called FLWOR expression, which stands for: `for`, `let`, `where`, `order by` and `return`. The `for` clause is like `from` in SQL, whereas the `let` is used to assign a result value to a variable. The clauses `where` and `order by` are similar to those in SQL. The results of the query are constructed as outputs by the `return` clause.

2.4.3 Advantages and Disadvantages of XML

XML Schema has more powerful data types and constraints, and it has the ability to model multi-valued and composite attributes, user-defined complex types, and cardinality for attributes and elements. The `key` and `keyref` are stronger than the `ID` and `IDREF` of DTD, since they are typed and can be composed. XML Schema offers flexible name space support, which is a significant advantage over using DTD [Valentine et al., 2002]. XML Schema allows the use of prefixed, default `namespaces` as well

as attributes and elements from other namespaces. Using XML Schema occurrence facility gives more flexibility than with DTD occurrence indicators. For example, an element can be modelled to have as least five occurrences using `minOccurs`="5". XML Schema is more expressive and more usable, compared to RDBs and DTDs. It provides a much more powerful means for defining document structures, and allows a much wider range of data types.

2.5 A Comparison of Data Model Concepts

Among the concepts offered by relational, object-based and XML databases, there are fundamental differences that result in data models' heterogeneity. Although they represent the same universe of discourse, data models are developed independently [Kappel et al., 2001]. In order to migrate a source database into another kind of database, the heterogeneity existing amongst data models, schemas and data need to be considered and resolved. Data model heterogeneity occurs due to differences between the concepts provided by an RDB on the one hand and those provided by object-based and XML on the other hand. Schema heterogeneity arises from the differences in the design goals of the database schemas [Kappel et al., 2001]. The aim of this section is to compare object-based and XML concepts against those of an RDB, concentrating on their standards: SQL2, ODMG 3.0, SQL3 and XML Schema, respectively. Placing homogeneous concepts together represents the foundation of the proposed CDM, based upon which the target databases can be generated.

2.5.1 Class Structure

The basic concepts of the relational data model are relation, tuple and attribute. Traditionally, a relation is shown as a table with columns (representing attributes), and the rows of the table represent tuples of the relation. Class structure is specified with a set of properties (attributes and relationships) and behaviours (methods). In an RDB model, real world entities are modelled in relations structurally, whereas the behaviours of these entities are indicated in application programs. However, object-based models combine properties and behaviours in a coherent structure, i.e., class. Objects of a class can be constructed and manipulated using methods supported by object systems, which are entirely different from the stored procedures in

RDBMSs. XML document structure is specified by element types and attributes, which are equivalent to relations and columns, respectively. Columns of a relation can be represented as attributes or sub-elements in the XML Schema. Relation tuples are equivalent to XML element instances. Elements are started and ended by tags, whereas data are assigned to attributes in RDB to form tuples. XML elements can be nested containing primitive or complex user-defined types, and relationships among them are modeled, forming a deep nesting hierarchy and references; whereas RDB data are flat, normalised and linked via foreign key constraints.

Attribute and user-defined data types: Each relation, class and element has a set of attributes, each of which has a data type. Attributes in RDBs are restricted to primitive data types such as integer and char, and tuples have a restricted data structure. Objects can have any data type supported by OO languages, which can be primitive or complex data types. Elements in XML documents have arbitrary orders using the `sequence` tag, however, they can be unordered using the `all` tag. As with object-based databases, XML Schema allows user-defined types. Unlike RDBs, a type can be defined based on other types in SQL4, ODMG 3.0 and XML Schema. User-defined types can be defined as named constructed types for reusing (e.g., named `row` or global `complexType`), or as anonymous to be defined without names inside their parent types (e.g., an anonymous `row` or local `complexType`).

Uniqueness, null and default values: Each RDB relation, object-based class and XML element has a unique name within the whole schema. An attribute name must be unique within its relation, class or element type. The name of an XML element and attribute is unique using the `namespaces` prefix [Fallside and Walmsley, 2004], which provides flexible naming and typing among elements without conflicts. XML Schema uses the "symbol space" to indicate a collection of names that are unique from each others. Thus, the same element name can be used many times in XML Schema if it uses a different name prefix. Attributes are allowed in SQL to have null and default values; however, XML Schema allows null and default values for attributes as well as elements. The SQL `default` clause is equivalent to the `DefaultValue` tag. Occurrence constraints in XML Schema are used to specify an element occurrence in the XML document. The `minOccurs` can be assigned to "0" for an element to accept nulls and to indicate that its instances occur none, one or more times, or it set to "1"

by default.

Identification: Tuples are identified in relations by means of primary key values, whereas the uniqueness of objects in object-based models is achieved using their OIDs. XML elements are accessed uniquely by the concept of ID in DTD. A primary key can be a single attribute or a composite of attributes, whereas ID is only a single attribute. However, XML Schema provides the key concept using the `key` keyword for single and composite keys. Primary keys in an RDB and XML Schema are value-based, whereas OIDs are object-based. The primary key concept is also supported in ODMG 3.0 and SQL4. Keys are not essential in object-based databases, because they are not used to implement relationships as is the case in RDBs. In object-based and relational models, primary keys are assigned by the database developers and their scope of identification is within a single relation/class, whereas the key scope in XML Schema is specified by XPath expression [Fallside and Walmsley, 2004].

2.5.2 Relationships

One of the main differences between diverse data models is how to handle relationships. Relationships in RDBs are represented by matching primary key and foreign key data. Referential integrity constraints are maintained by mean of foreign keys. For each foreign key value there exists a matched primary key value, which can be considered as a value reference. In contrast, object-based databases store OIDs within objects to indicate other objects to which they are related. Similar to RDBs, primary key and foreign key concepts are provided by XML Schema for relationship definitions using the `key` and `keyref` tags, respectively. However, similar to object-based data models, relationships can be represented in XML Schema by specifying nested complex types and inheritance hierarchies. Elements can be defined as components or sub-classes under other complex elements. Relationship types may be associations, aggregations and inheritances.

Association: Similar to RDBs, using `key` and `keyref`, a particular key of an XML element can be referenced by a corresponding foreign key. Object-based databases allow associations to be defined between objects uni-directionally and bi-directionally using OIDs. Objects participate in associations explicitly by placing the OIDs of

related objects within the objects themselves, maintaining referential integrity. Associations are implemented using OIDs equivalent to foreign keys in RDBs. However, in RDBs the primary key is located in the M side as a foreign key; whereas the OID might be located as a collection on the 1 side. As RDB attributes are limited to being single values, the M:N association has to be handled in a separate relation. Object models and the XML Schema allow multi-valued attributes, and hence M:N relationships can be directly modeled. However, M:N relationships are difficult to represent in XML documents [Elmasri and Navathe, 2006]. Collections are used to store multiple values in a single attribute in object-based databases, whereas the `maxOccurs` attribute specifies the maximum occurrences of an element, e.g., `maxOccurs`= `unbounded`.

Aggregation: In a relational model, a relation consists of a set of tuples of atomic values only. A weak entity, which represents a component of another entity (whole-part relationship), can be identified using data dependency via foreign keys. However, a complex object can be composed from other objects (that may be composite themselves) using certain types of constructors such as `set`, `list` and `bag`. In contrast, XML allows any level of nesting in which an element can contain other elements forming a component (part-of) element, which is similar to a composite type in object data models. Elements in XML can be primitive type elements, composite elements which contain only one another element, or composite elements with mixtures of primitive element types and other elements. An element that does not contain primitive or composite elements is called an `empty` element.

Inheritance: A relational data model does not support an inheritance concept directly; however, a variety of alternatives can be adapted to extract and represent this type of relationship [Akoka et al., 1999; Elmasri and Navathe, 2006]. In contrast, inheritance is very important in object-based and XML models. It allows better and more accurate descriptions of reality to be achieved since it organises object-based classes/types in a hierarchy for specifications sharing and reusing. A sub-class types inherits properties and methods of a pre-defined super-class, adding its own new properties and methods. This can be achieved using built in inheritance constructs, such as '`:`' and `extends` in an ODMG 3.0, and `under` in SQL4. XML Schema allows new element types to be extended based on pre-defined types using the `extension` and `restriction` mechanism. However, ODMG 3.0, SQL4 and XML Schema do not

support multiple inheritance among concrete types.

2.6 Summary

This chapter has given a brief overview of the main concepts of relational, object-based and XML data models. The chapter started by introducing the basic concepts of the models, and then the standards they support as well as the advantages and disadvantages of each data model. Section 2.1 introduced the essential concepts of relations, attributes, tuples and constraints. Then, the SQL2 standard and the limitations of RDBMS were stated. Object-based models were then discussed in Sections 2.2 and 2.3. Section 2.2 described the essential concepts used in OODBs, including ADTs, data encapsulation, inheritance, polymorphism, and classes and complex objects. The section then focused on the ODMG 3.0 object model. The object-relational model and recent features of the SQL3 and SQL4 standards were introduced in Section 2.3, whereas Section 2.4 presented an XML data model in the context of databases. XML concepts were discussed briefly, focusing on the XML Schema language. Diverse aspects, including data models, structure, typing mechanism, and relationships were explained. A comparison of the concepts of object-based and XML databases against those of an RDB was given in Section 2.5. Similarities among the diverse data models produce natural correspondences, which are exploited to bridge the semantic gap between their concepts, providing the basis for the CDM, which is used as an intermediate representation to migrate an RDB into the target databases.

Chapter 3 provides a survey of the recent literature about various research trends relevant to the migration of RDBs. A detailed comparison is given of selected work, involving the approaches and techniques used to solve the problem of migrating an RDB into an OODB, ORDB and XML.

Chapter 3

Relational Database Migration Approaches

The solution proposed in this dissertation is a method for migrating a source RDB into an OODB, ORDB XML as targets. Chapter 2 has provided background information about the source and target data models, including database systems and current standards. This chapter provides an investigation into the problems of RDB migration. It reviews various techniques and proposals for this purpose, identifies their differences and assesses the impact of existing literature and shows how it has shaped current and future research in this area. We focus on the case where the input is an RDB and the outputs are OODB, ORDB and XML. Hence, the reverse process (e.g., migrating OODBs into RDBs) are not covered. A summarised form of this chapter has recently been published [Maatuk et al., 2008b].

The remainder of this chapter is organised as follows. Section 3.1 reviews current approaches and techniques related to database conversion. Section 3.2 gives an overview of proposals for RDB migration. Section 3.3 presents a review of existing proposals for migrating RDBs into OODBs, and work on mapping RDBs into ORDBs is reviewed in Section 3.4. Section 3.5 then provides a review of work on migrating RDBs into XML. Section 3.6 concludes the chapter, whereas Section 3.7 provides a summary of the chapter and points to what follows next in the dissertation.

3.1 Approaches and Techniques

This section introduces approaches and techniques related to database conversion. Section 3.1.1 discusses approaches to database conversion whereas Section 3.1.2 discusses existing translation techniques.

3.1.1 Conversion Approaches

There are three approaches related to database conversion. The first approach is for handling data stored in RDBs through OO/XML interfaces. Connecting an existing RDB to a conceptually different database system is the basis of the second approach, and the third approach is to migrate an RDB into a target database. The first and second approaches deal with schema translation, whereas in the third approach both schema and data are completely migrated into a target database. The requirements of database systems determine which approach is most suitable to adopt [Behm, 2001; Henrard et al., 2002]. Due to substantial investments in many traditional RDBs, part of their data may need to be formatted and implemented in a new and different platform. Hence, constructing a gateway interface between the two databases might be preferred. Migrating to a new DBMS might be a good decision to make if the existing system is too expensive to maintain.

Approach 1: Non-relational applications on top of RDBs

Data may be required to be processed in object/XML form and stored in relational form based on the concept of object for programs and RDB for persistence. This process requires object-to/from-relational and XML-to/from-relational mapping techniques, which link RDBs to non-relational applications. This approach is also known as wrappers [Malki et al., 2001]. Such mapping is bi-directional on demand of updating an RDB using OO/XML interfaces. This is the reverse direction from where object-based/XML schemas are translated into an RDB schema.

Viewing objects on top of RDBs: While OO objects are associated via references, data in RDB tables are linked through the values of primary keys and foreign keys. A single object might be represented by several tuples in several tables, and therefore, joining these tables is required for queries. The problem lies in converting

these objects to tabular forms in order for them to be stored in and retrieved from RDB systems when needed. This constant conversion leads to a semantic gap between the two different paradigms, which is known as the object-relational impedance mismatch [Hohenstein, 1996; Ambler, 2003]. To avoid this, developers have to write large amounts of code to map objects in programs into tuples in an RDB, which can be very time-consuming to write and execute. Another solution would be to use mapping query systems/middleware. Query systems, e.g., Penguin [Keller and Wiederhold, 2001] support object views for RDBs, which enable non-traditional applications to share data with their object schema [Takahashi and Keller, 1993, 1994]. Penguin is an ODBMS that relies on RDBs for persistence [Keller and Wiederhold, 2001]. Middleware is a software that links OOPL concepts to data stored in RDBs through ODBC/JDBC, thus creating a virtual object database. JDBC is a set of Java classes for specific databases interacting with ODBC [Hamilton et al., 1997]. Other modern systems include Java data object (JDO) [JDO, 2009], Rogue Wave [Roguewave, 2006] and Oracle TopLink [TopLink, 2006]. These provide mapping tools for binding tuples in RDBs, making them appear as objects for OOPLs. However, mapping using middleware requires time for schema mapping, on each occasion that stored data are accessed.

Publishing relational data as XML documents: RDB data can be published as XML documents, using special declarative languages, to be exchanged over the Web. Various proposals, which make RDB data accessible to XML have been described [Turau, 1999; Carey et al., 2000a; Fernandez et al., 2000; Funderburk et al., 2002; Shanmugasundaram et al., 2001; Liu et al., 2003; Duta et al., 2004; Chebotko et al., 2007]. Through converting an RDB into XML, users see views that can be queried using XML query languages. That is, a user can pose queries on XML views, which are in turn translated into SQL queries to derive the required results from an RDB. However, data in such applications is not fully materialised in XML form, whereas the results are. Furthermore, adapting the object view for representing XML data in an RDB faces restrictions, such as data collection representations and tag naming. In DTD schema it is not possible to utilise collections within collections, and providing XML elements based on column names may result in tag name conflicts [Valikov et al., 2001]. SilkRoute [Fernandez et al., 2000, 2001], XPERANTO [Carey et al., 2000a], XTABLES [Funderburk et al., 2002] and VXE-R [Liu et al., 2003] are among

the systems taking this approach.

Fernandez et al. [2000, 2001] proposed a tool called SilkRoute for mapping relational data into XML virtual views using a declarative query language (i.e., RXL), and querying these views using another language called XML-QL. However, the tool cannot store or query XML documents. The XPERANTO system translates XML-based queries into SQL over (object-)RDBs [Carey et al., 2000a,b; Shanmugasundaram et al., 2001]. The system receives and de-constructs SQL queries and returns XML documents. However, users have to specify the queries and define more complex views using an appropriate query language, once the system publishes a default XML view. In addition, mismatches exist between XML and SQL query syntax, and more advanced object features and integrity constraints are not considered precisely. Unlike SilkRoute, XTABLES provides the user with a single query language which can be used to query seamlessly over relational data and metadata [Funderburk et al., 2002]. In addition, XTABLES can query and store XML documents in RDBs. VXE-R is an engine for translating relational schema into an equivalent XML schema, where XML queries can be issued directly against XML schema [Liu et al., 2003]. Finally, DB2XML is a tool for converting RDBs into DTD documents [Turau, 1999].

Storing XML documents in RDBs: XML documents can be stored in RDBMSs [Yoshikawa et al., 2001; Bourret, 2005; Fong et al., 2006]. A database that allows XML data to be stored in it is called an XML-enabled database. A whole document can be stored in a large single column in a table. The column can be a BLOB or a CLOB. A key column can be added to manage this table. A more complex technique is to shred a document into elements and store each element in a separate table. It models pure data without a document and has the flattened form of a relational model [Bourret, 2005]. XML-enabled databases are useful for retrieving and storing data which conform to XML form. However, they cannot effectively store a complete document with its identity, order and comments. Therefore, it may be preferable to handle such documents with native-XML databases, because they use XML models that can store whole documents directly without requiring any mapping.

Approach 2: Database Integration

A connection can be established between RDBs and other databases which allows the applications built on top of a new DBMS to access both relational and object/XML

DBMSs, giving the impression that all data are stored in one database. This represents a simple level of database integration between systems [Tari and Stokes, 1997; Tari et al., 1997; Parent and Spaccapietra, 2000; Collins et al., 2002]. This is achieved using a special type of software called *gateways*, which support connectivity between DBMSs and do not involve the user in SQL and RDB schema. Hence, queries and operations are converted into SQL and the results are translated into target objects [Orenstein and Kamber, 1995]. Many applications use two or more underlying databases. On retrieving data from both systems, the unification of their two schemas is necessary by providing two-way mapping. During integration, systems cooperate autonomously by creating a unified and consistent data view for several databases, hiding heterogeneities and query languages [Hohenstein and Plesser, 1996]. Most commercial DBMSs such as Objectivity [Objectivity, 2009] and ObjectStore [ObjectStore, 2009] provide flexibility of mapping and gateways construction among heterogeneous databases. The difference between gateways and object-relational mapping tools is that, in the former, objects are persistently stored in the new developed database system; whereas in mapping or publishing data, objects are created and handled in the normal way but are stored in an RDB. However, in both approaches old data stored in an RDB are retained.

Approach 3: Database Migration

Migration of an RDB into its equivalents is usually accomplished between two databases according the literature. The first database is an RDB, called the *source*, and the second, called the *target*, which represents the result of the migration process. In addition, the process is performed with or without the help of an intermediate conceptual representation, e.g., an ER model as a stage of enrichment. The input source schema is enriched semantically and translated into a target schema. Data stored in the source database are converted into the target database based on the target schema. Generally, relations and attributes are translated into equivalent target objects. Foreign keys may be replaced by another domain or relationship attributes. Weak entity relations may be mapped into component classes, multi-valued or composite attributes inside their parent class/entity. Other relationships, such as associations and inheritance, can also be extracted by analysing data dependencies or database instances. In data conversion, attributes that are not foreign keys become literal attribute values of objects, elements or sets of elements. Foreign keys realise relationships among tuples,

which are converted into value-based or object references in a target database. The challenge in this process is that the data of one relation may be converted into a collection of literal/references rather than into one corresponding type. This is because of the heterogeneity of concepts and structures in the source and target data models.

3.1.2 Translation Techniques

Existing techniques used for RDB schema translation can be classified into two types: (i) source-to-target (S2T), including flat, clustering and nesting translation techniques, and (ii) source-to-conceptual-to-target (SCT) translation. In some of these techniques, data might be converted based on the resulting target schema.

Source-to-target (S2T) technique

This type of technique translates a physical schema source code directly into an equivalent target. However, as the target schema is generated using one-step mapping with no intermediate stage for enrichment, this technique usually results in an ill-designed database because some of the data semantics (e.g., integrity constraints) are not considered. This approach can take the following three forms:

Flat technique: This technique converts each relation into an object class/XML element in the target database [Premerlani and Blaha, 1994; Fong, 1997; Wang, 2004; Wang et al., 2005]. Foreign keys are mapped into references to connect objects. However, due to the one-to-one mapping, the flattened form of RDBs is preserved in the generated database, so that object-based model features and the hierarchical form of the XML model are not exploited. This means that the target database is semantically weaker and of a poorer quality than the source. Moreover, creating too many references causes degraded performance during data retrieval.

Clustering technique: This technique is performed recursively by grouping entities and relationships together starting from atomic entities to construct more complex entities until the desired level of abstraction (e.g., aggregation) is achieved [Getta, 1993; Yan and Ling, 1993; Missaoui et al., 1995; Sousa et al., 2002]. A strong entity is wrapped with all of its direct weak entities, forming a complex cluster labelled with the strong entity name. This technique works well when the aim is to produce hierarchical forms with one root. This technique may reduce search time by avoiding join operations, and thus speeding up query processing, however, it may lead to

complex structures and is prone to errors in translation. In addition, materialising component entities within their parent/whole entities may cause data redundancy, the loss of semantics and the breaking of relationships among objects.

Nesting technique: This technique uses the iterated mechanism of a `nest` operator to generate a nested target structure from tuples of an input relation [Lee et al., 2002; Singh et al., 2004]. The target type is extracted from the best possible nesting outcome. For a table T with a set of columns X, nesting on a non-empty column(s) $Y \in X$ collects all tuples that agree on the remaining columns $X - Y$ into a set [Lee et al., 2002]. However, the technique has various limitations, e.g., mapping each table separately and ignoring integrity constraints. Besides, the process is quite expensive, since it needs all tuples of a table to be scanned repeatedly in order to achieve the best possible nesting.

Source-to-conceptual-to-target (SCT) technique

This type of technique enriches a source schema by data semantics that might not have been clearly expressed. The schema is translated from a logical into a conceptual schema through recovering the domain semantics (e.g., primary keys, foreign keys, cardinalities, etc.) and making them explicit. The results are represented as a conceptual schema using database reverse engineering (DBRE) [Chiang et al., 1994]. The resulting conceptual schema can be translated into the target logical schema effectively using database forward engineering (DBFE). In this way, the technique results in a well-designed target database.

Database reverse engineering (DBRE): DBRE is a process for enriching a source schema using semantics that might have not been clearly expressed by acquiring as much information as possible about objects and the relationships that exist among them [Castellanos et al., 1994]. This process is also known as semantic enrichment. Inferring conceptual schema from a logical RDB schema via DBRE has been extensively studied [Hainaut, 1991; Andersson, 1994; Chiang et al., 1994; Hainaut et al., 1997; Malki et al., 2002; Alhajj, 2003]. Such conversions are usually specified by rules, which describe how to derive RDB constructs (e.g., relations, attributes, data dependencies, keys), classify them, and identify relationships among them. Semantic information is extracted by an in-depth analysis of relations in an RDB schema together with their data dependencies into a conceptual schema such as ER, UML,

OO and XML data models. Data and query statements have also been used in some studies to extract data semantics. Some proposals consult expert users or use data dictionaries to provide metadata, whereas other proposals employ database design techniques. However, some of these proposals could be combined together to form a more comprehensive solution. Table 3.1 shows a summary of some of these proposals, details on which are given below.

Proposal	Schema	Data	DDL, DML	data dictionary	Expert User	Design
[Chiang et al., 1994]	√	√	×	×	√	×
[Fonkam and Gray, 1992]	√	×	×	×	×	×
[Alhajj, 2003]	×	√	×	√	√	×
[Soutou, 1996]	×	√	√	×	×	×
[Petit et al., 1994]	×	√	√	×	×	×
[Andersson, 1994]	×	×	√	×	×	×
[Soutou, 1998a]	√	√	×	√	×	×
[Alhajj and Polat, 2001]	×	√	×	√	√	×
[Hainaut et al., 1994]	√	×	×	×	×	√
[Marcos et al., 2003]	√	×	×	×	×	√
[Tari et al., 1997]	√	√	×	×	×	×
[Malki et al., 2002]	√	√	×	×	√	×

√: Yes ×: No

Table 3.1: Extracting an RDB conceptual schema via DBRE

- **Schema-based proposals:** Most of the existing DBRE studies fall into this category, where the inputs are RDB schemas and the outputs are data semantics from analysing relations and attributes [Navathe and Awong, 1988; Davis and Arora, 1988; Johannesson and Kalman, 1989; Fonkam and Gray, 1992; Chiang et al., 1994; Johannesson, 1994]. The extraction of data semantics by converting an RDB schema into an EER model has been studied in the early nineties [Fonkam and Gray, 1992; Chiang et al., 1994]. Three algorithms are proposed to extract a conceptual ER from an existing RDB based of the classification of relations and attributes [Navathe and Awong, 1988; Davis and Arora, 1988; Johannesson and Kalman, 1989]. However, all those algorithms do not consider inheritance relationships. Fonkam and Gray [1992] presented a more general algorithm that is based on these algorithms, where the original contribution of this algorithm was to establish generalisation hierarchies. Chiang et al. [1993, 1994] proposed a method that focuses on deriving an EER from a 3NF RDB. This type of method uses a variety of heuristics to recover domain semantics through the classification of relations, attributes and key-based inclusion dependencies using the schema. However, expert involvement is required to distinguish between similar EER constructs, i.e., weak entities and specific

relationship types [Chiang et al., 1994]. In addition, the consistency of key naming and a well-formed schema is assumed.

- **Data content-based proposals:** Several studies have proposed the extraction of semantics by analysing data instances and possibly schemas [Chiang et al., 1994; Soutou, 1996; Tari et al., 1997; Alhajj, 1999, 2003]. Soutou [1996] proposed a process for extracting the cardinalities of n-ary relations representing relationships by generating a set of SQL queries. Data instances are used for relation classifications with respect to their keys [Chiang et al., 1994; Tari et al., 1997]. Alhajj [2003] developed algorithms that utilise data to derive all possible candidate keys for identifying the foreign keys of each given relation in a legacy RDB. This information is then used to derive a graph called RID, which includes all possible relationships among RDB relations. The RID graph works as a conceptual schema [Alhajj, 1999].

- **Query-based proposals:** Inferring a conceptual schema based on the analysis of DDL and SQL queries embedded in applications has been suggested by several authors [Andersson, 1994; Petit et al., 1994; Comyn-Wattiau and Akoka, 1996; Akoka et al., 1999]. Petit et al. [1994] presented a method to extract EER model constructs from an RDB by analysing SQL queries in application programs. In common with Andersson [1994], Petit et al. [1996] extracted a conceptual schema by investigating equi-join statements. The method uses a join condition and the `distinct` keyword for attribute elimination during key identification. Comyn-Wattiau and Akoka [1996] proposed a method called MeRCI which concentrates on schema de-optimisation. The process starts with RDB physical schema containing de-normalised relations, and then a set of appropriate rules is applied to de-optimize the schema through the analysis of application source codes (DDL, DML) and data mining techniques. Relational operators such as `join`, `project` and `restrict` in a physical schema are detected and used for de-normalisation of relations. Akoka et al. [1999] focused on extracting generalisation hierarchies in an RDB using DDL, DML and data analysis.

- **Other proposals:** Soutou [1998a,b] presented an algorithm for inferring n-ary relationships from RDBs using a combination of data dictionary, and the analysis of schema and data. Alhajj and Polat [2001] re-engineered an RDB

into an OODB using an expert user and the data dictionary as primary sources of information. Since an RDB does not enable a natural way of representing inheritances, several heuristic and algorithmic methods have been proposed to elicit inheritance relationships hidden in RDBs [Fonkam and Gray, 1992; Akoka et al., 1999; Lammari, 1999; Al-Kamha et al., 2005; Elmasri and Navathe, 2006; Lammari et al., 2007]. Data instances, schemas, DDL and DML specifications, along with understanding null value semantics, are used to detect inheritance.

- **Design-based proposals:** Some works that have design characteristics can be used for DBRE [Hainaut et al., 1993; Getta, 1993; Hainaut et al., 1994; Marcos et al., 2003]. A method based on a generic schema specification model and DBRE techniques has been proposed to deal with design and re-engineering database applications [Hainaut et al., 1994]. Marcos et al. [2003] presented rules to translate a UML class diagram into an ORDB schema in SQL3 and Oracle 8_i.

  The problems of semantic enrichment arise from processing badly-designed and poorly documented applications [Hainaut, 1991]. Many RDBs might have been specified without definition of constraints, such as keys and integrity constraints [Behm et al., 2000]. These semantics specified into conceptual schema might not be presented explicitly in data dictionaries [Pérez et al., 2003]. For example, foreign keys are not possible in Oracle 5. Moreover, many RDBs do not contain semantic constraints for optimisation reasons, and not all databases are built by experienced developers, who may produce poor or inadequate structures [Hainaut, 1991].

Database forward engineering (DBFE): This process is known as schema translation. A conceptual schema generated from the DBRE process can be translated into a high level data model through the application of a set of rules, called schema mapping rules. Several proposals have been made for transforming conceptual schemas, e.g., ER, EER, UML or other specific models into object-based and XML schemas [Narasimhan et al., 1993; Hainaut et al., 1994; Orenstein and Kamber, 1995; Carey et al., 2000a; Du et al., 2001; Kleiner and Lipeck, 2001; Marcos et al., 2003; Vela and Marcos, 2003]. These proposals and many others have been used as a basis for middlewares, gateways and CASE tools. A review of database design transformations based on the ER model may be found [Fahrner and Vossen, 1995a].

3.2 RBD Migration Proposals

This section presents a number of properties of existing proposals for RDB migrations and tools used to facilitate the migration process. Those properties are discussed in Section 3.2.1, whereas migrations tools are presented in Section 3.2.2.

3.2.1 Database Migration Properties

Before we embark on a detailed review on proposals used in an RDB migration, this section describes a set of properties which can be used to compare and evaluate existing proposals. Indeed, each proposal has its properties, e.g., prerequisites and data model used. These properties lead to different mapping rules for the migration process, which in turn affect the results and quality of the process. Table 3.2 provides a comparison and classification of some of these proposals showing the input and target generated databases, and technique used and prerequisites of each proposal. In addition, Table 3.3 surveys such proposals, showing intermediate conceptual models used (if any), the preservation of data semantics and some other features. These properties surveyed in both tables are explained below. However, detailed descriptions on these proposals as works for migrating RDBs into OODBs, ORDBs and XML according to these properties are given in Sections 3.3, 3.4 and 3.5, respectively.

Proposal	ST	DC	Tec	Input	Prerequisites	Output		
						OODB	ORDB	XML
[Fong, 1997]	√	√	S2T	RDB	FD,ID,ED	√	×	×
[Yan and Ling, 1993]	√	×	S2T	RDB	keys, ID	√	×	×
[Ramanathan and Hodges, 1997]	√	×	S2T	RDB	FD, PKs, FKs, 2NF	√	×	×
[Zhang et al., 1999]	√	×	S2T	RDB	FD, ID, 4NF, MVD	√	×	×
[Fahrner and Vossen, 1995b]	√	×	S2T	RDB	keys, FD, ID, 3NF	√	×	×
[Behm et al., 2000]	√	√	SCT	RDB	keys, DD, Ins	√	×	×
[Alhajj and Polat, 2001]	√	√	SCT	RDB	keys, DD, Ins	√	×	×
[Narasimhan et al., 1993]	√	×	S2T	ER	ER	√	×	×
[Premerlani and Blaha, 1994]	√	×	S2T	RDB	keys, non-3NF	√	×	×
[Castellanos et al., 1994]	√	×	S2T	RDB	FD, ID, ED, non-3NF	√	×	×
[Urban et al., 2001]	√	×	S2T	UML	UML class diagram	×	√	×
[Marcos et al., 2003]	√	×	S2T	UML	UML class diagram	×	√	×
[Arora et al., 2005]	√	×	SCT	RDB	ER, UML	×	√	×
[Vela and Marcos, 2003]	√	×	S2T	UML	UML class diagram	×	√	×
[Fong and Cheung, 2005]	√	√	SCT	RDB	PKs, FKs	×	×	√
[Kleiner and Lipeck, 2001]	√	√	S2T	EER	FD, ID	×	×	√
[Du et al., 2001]	√	×	SCT	RDB	3NF	×	×	√
[Fong et al., 2003]	√	√	SCT	RDB	FD, MVD, JD, TD	×	×	√
[Lee et al., 2002]	√	×	S2T	RDB	PKs, FKs	×	×	√
[Wang et al., 2005]	√	√	SCT	RDB	PKs, FKs, DD	×	×	√
[Laforest and Boumediene, 2003]	√	×	S2T	RDB	PKs, FKs	×	×	√
[Fong et al., 2006]	√	√	SCT	RDB	keys, FD, IN, MVD	×	×	√

ST: Schema Translation DC: Data Conversion Tec: Technique FD: Functional ID: Inclusion Dependency MVD: Multi-valued Dependency TD: Transitive Dependency Ins: Data instances Dependency ED: Exclusion Dependency JD: Join Dependency PK: primary key FK: foreign key DD: data dictionary

Table 3.2: RDB migration (prerequisites, input and output databases)

Migration prerequisites: Existing work on database migration enforces different prerequisites on the source databases being migrated. These include the consistency of naming attributes, the availability of all keys and schema, inclusion and functional dependencies, and database instances. Most existing proposals are limited by the assumptions that they make. For instance, a source schema is required to be available for further normalisation to third normal form (3NF) [Chiang et al., 1994; Fahrner and Vossen, 1995b] or even to 4NF [Zhang et al., 1999] before the migration process can begin. However, this is not a practical choice for existing RDBs. Data dependency, which is most often represented by key constraints, plays the most important role in this process. Evaluation of functional, inclusion and key-based dependency is assumed in many proposals. Other kinds of data dependency may also be required, e.g., multi-valued dependency (MVD) [Zhang et al., 1999; Fong et al., 2003] and exclusion dependency (ED) [Castellanos et al., 1994; Fong, 1997]. Premerlani and Blaha [1994] assume that the problem of synonyms and homonyms has been resolved prior to database migration. Also, the classification of relations with respect to their keys, e.g., to know whether the primary keys and foreign keys are constructed from each other may be required [Chiang et al., 1994; Tari et al., 1997]. Other frequent assumptions are that the initial schema is well-designed and that all basic relevant constraints are given in the descriptions of the schema or provided by the user [Behm et al., 2000; Alhajj, 2003; Wang et al., 2005].

Input and output models: In existing work, the RDB migration process usually takes one RDB as input and aims to generate one target database. A source schema is translated into another equivalent schema and data are converted in accordance with schema translation. However, most work to date has focused on translating RDB schemas directly into schemas of other non-standardised data models, in the context of database integration [Castellanos et al., 1994; Fong, 1997; Zhang et al., 1999]. Few attempts have been made to generate target data models based on their conceptual schemas or other representations, as an intermediate stage for enrichment. Numerous methods have been proposed for DBRE by transforming logical data models into ER, EER and UML models. A large body of literature exists on DBFE (or database design) aiming to transform such conceptual models to logical data models. In addition, only few works consider current standards, i.e., ODMG 3.0, SQL4 and XML Schema as target models [Fahrner and Vossen, 1995b; Wang et al., 2005].

Proposal	ICR	Data Semantics					Features		Target
		AS	AG	IN	RI	OP	SA	UI	model
[Fong, 1997]	-	√	×	√	×	×	×	H	NS
[Yan and Ling, 1993]	-	√	√	√	×	×	×	H	NS
[Ramanathan and Hodges, 1997]	-	√	√	√	×	√	×	H	OMT
[Zhang et al., 1999]	-	√	√	√	×	×	×	H	NS
[Fahrner and Vossen, 1995b]	-	√	√	√	√	√	√	H	ODMG-93
[Behm et al., 2000]	SOT	√	×	√	√	×	√	L	ODMG-93
[Alhajj and Polat, 2001]	RID	√	√	√	×	×	×	L	NS
[Narasimhan et al., 1993]	-	√	√	×	×	×	×	H	NS
[Premerlani and Blaha, 1994]	-	√	√	√	×	√	×	H	OMT
[Castellanos et al., 1994]	-	√	√	√	×	×	×	H	BLOOM
[Urban et al., 2001]	-	√	√	×	√	√	×	H	Oracle 8_i
[Marcos et al., 2003]	-	√	√	√	√	√	√	H	SQL3
[Arora et al., 2005]	UML	√	√	√	×	×	×	H	Oracle 9_i
[Vela and Marcos, 2003]	-	√	√	√	√	×	√	H	Oracle 8_i
[Fong and Cheung, 2005]	EER	√	√	√	√	×	√	L	XML Schema
[Kleiner and Lipeck, 2001]	-	√	×	×	×	×	√	H	DTD
[Du et al., 2001]	ORA-SS	√	√	√	√	×	√	H	XML Schema
[Fong et al., 2003]	DOMs	√	√	×	×	√	√	L	DTD
[Lee et al., 2002]	-	√	√	×	×	√	√	H	DTD
[Wang et al., 2005]	ER	√	×	×	√	×	√	H	XML Schema
[Laforest and Boumediene, 2003]	-	√	√	×	×	×	√	H	DTD
[Fong et al., 2006]	EER	√	√	√	×	×	√	H	DTD

ICR: Intermediate Conceptual Representation AS: Association AG: Aggregation IN: Inheritance RI: Referential Integrity OP: Optimisation SA: Standard Adoption UI: User Interaction L: Low intervention H: High intervention NS: Non-standard

Table 3.3: RDB migration (data semantics and features)

Conceptual models used: Earlier models such as ER, EER and object-modeling technique (OMT) [Rumbaugh et al., 1990] are assumed in most studies as a conceptual model or target data models, whereas other works are restricted to a particular product, e.g., Oracle [Arora et al., 2005]. To enrich a source RDB structurally and semantically, graphs and models are proposed as an intermediate stage [Abelló et al., 1999; Dobbie et al., 2000; Du et al., 2001; Alhajj, 2003; Fong and Cheung, 2005]. A graph called an RID, developed by Alhajj [2003], has been used to translate an RDB into an OODB [Alhajj and Polat, 2001] or into an XML [Wang et al., 2005]. This graph, similar to an ER diagram is used for identifying relationships and cardinalities. A model, called Barcelona object oriented model (BLOOM), has also been developed to act like a canonical model for federated DBMSs [Abelló et al., 1999; Abelló and Rodríguez, 2000]. Its main goal is to upgrade the semantic level of the local schemas of different databases and to facilitate their integration. Behm et al. [2000] proposed a model, called semi object type (SOT), to facilitate the restructuring of schemas during the translation of an RDB into an OODB. Another model, called ORA-SS, has been proposed to support the design of non-redundant storage of semi-structured data models [Dobbie et al., 2000]. The ORA-SS is used as an intermediate model to map an RDB into an XML Schema [Dobbie et al., 2000]. The model has its own diagrammatic notations for expressing class attributes and relationships, similar to those of ER and OO data models. The model represents data as directed graphs, and focuses on modelling n-ary relationships as well as distinguishing between the

attributes of relationships and those of objects. However, it uses the technique of nesting and referencing in representing relationships among objects.

Semantic preservation: RDBs typically contain implicit and explicit data semantics, concerning integrity constraints and relationships among relations. Target databases should hold equivalents to these semantics. Several previous proposals have failed to explicitly maintain all of the data semantics (e.g., integrity constraints and inheritance). Constraints are instead mapped into class methods [Fahrner and Vossen, 1995b] or into separate constraint classes [Narasimhan et al., 1993]. Relationships are translated in most of the work, however, inheritance relationships have not been fully addressed. Few studies address database optimisation issues, e.g., horizontal and vertical partitioning [Narasimhan et al., 1993; Ramanathan and Hodges, 1997]. Object-based data models consist of static properties (attributes and relationships) and dynamic properties (methods or functions), which make them richer than relational data models. Most existing methods focus on constructing a static rather than dynamic target schema.

User involvement: A common observation in the different proposals is that user interaction is necessary at some point to provide additional information to achieve the desired results. User intervention might be required for the classification and understanding of keys in an RDB [Castellanos et al., 1994], choosing the appropriate transformation rule [Jahnke et al., 1996; Behm et al., 1997], or identifying complex relationship structures [Fahrner and Vossen, 1995b]. User involvement is also required for resolving optimisation issues such as naming conflicts and vertically or horizontally partitioned relations [Chiang et al., 1994; Premerlani and Blaha, 1994; Ramanathan and Hodges, 1996], and for selecting XML documents' roots and directing the conversion process [Lee et al., 2002; Fong et al., 2003].

3.2.2 Tools Support

A number of prototypes and tools have been developed to facilitate the migration of RDBs into target databases [Chiang, 1995; Jahnke et al., 1996; Monk et al., 1996; Amer-Yahia, 1997; Turau, 1999; Lo et al., 2004; Wang et al., 2005].

Chiang [1995] presented a system, called the knowledge extraction system (KES), for generating an EER model from RDBs. KES has been developed to extract domain semantics by analysing the RDB schema and data instances. Monk et al. [1996] proposed a tool for transforming an RDB into an OODB, where a schema and data are created using convertors which can then be exploited by client programs using translators. However, various semantic constraints, schema-mapping constructs and data migration techniques were not addressed adequately in this work.

Jahnke et al. [1996]; Jahnke and Zundorf [1998] described a semi-automatic tool for mapping an RDB into an ODMG ODL schema. The conversion process is provided by an adapted set of schema mapping rules to produce an initial OO conceptual schema. Once the OO schema is produced, it can be refined to exploit OO concepts, e.g., inheritance and aggregation using the *Varlet* redesign tool. This tool is similar to the RELS tool [Pérez et al., 2003]. However, *Varlet* focuses on migrating legacy databases, which are enriched with semantic information inferred using other techniques (i.e., [Premerlani and Blaha, 1994; Fahrner and Vossen, 1995b]), whereas any missing semantics information is provided by an expert user.

Amer-Yahia [1997]; Jahnke and Zundorf [1998] provided a tool for RDB-OODB mapping with an independent language called *RelOO*. Data conversion is performed in more than one transaction according to three criteria: a) a certain number of objects are mapped in one transaction, b) each relation is fragmented into several partitions during its mapping into an object class, and c) a period of time is assigned within which each transaction is finished. However, the tool does not exploit all of the features provided by the OODB paradigm, such as inverse references, inheritance and aggregation, and the fragmentation of tables during conversion might cause unnecessary complexity.

A system named conversion of catalog-based and legacy RDBs to XML (COCALEREX) has been developed to convert RDBs into XML documents [Wang et al., 2004; Wang, 2004; Wang et al., 2005]. Similar tools are VIREX [Lo et al., 2004] and Conv2XML [Duta et al., 2004]. COCALERTEX can convert legacy and catalog-based RDBs, whereas Conv2XML assumes that basic constraints, e.g., primary keys, foreign keys, unique keys and nulls are already available, and the schema must be in 3NF.

3.3 Migrating RBD into OODB

An overview of the main concepts for migrating RDBs into object-based and XML databases has been provided in Section 3.2. In this section, existing proposals for the migration of RDBs into OODBs are discussed in further detail, including semantic enrichment, schema translation and data migration.

ER-to-OODB: Narasimhan et al. [1993] proposed a procedure that deals with an RDB abstraction through mapping its related ER model into an OO schema to exploit the ER model features, e.g., multi-valued attributes. The work suggests creating a separate constraint class with methods as a sub-class for each of the OODB classes. The translation of EER models into OO models by a set of transforation rules has been illustrated [Getta, 1993; Fong, 1995]. Whereas Fong [1995] mapped an EER model into an OMT model, Getta [1993] used a specific OO model as a target. In Fong's algorithm, EER strong entities are mapped into classes with corresponding attributes [Fong, 1995]. Weak entities and aggregations are mapped into component and composite object-classes, respectively. Relationships among entities are mapped into associations, generalisation/specialisation into inheritance, and categorisations into multiple-inheritance. Other methods of transforming ER, EER and UML models into OODBs in the context of database design have been reported [Fahrner and Vossen, 1995a; Cattell and Barry, 2000; Date, 2002; Connolly and Begg, 2002; Elmasri and Navathe, 2006; Garcia-Molina et al., 2008].

RDB-to-OODB: Several methods have been proposed for migrating RDBs into OODBs directly, i.e., without using an intermediate conceptual representation [Castellanos et al., 1994; Premerlani and Blaha, 1994; Fahrner and Vossen, 1995b; Fong, 1997; Ramanathan and Hodges, 1997; Zhang et al., 1999]. However, all these proposals, except Fong [1997], concern only schema translation.

Premerlani and Blaha [1994] proposed a procedure for mapping an RDB schema into an OMT schema. An OMT schema is produced by representing each RDB relation with its attributes as an OMT class, and primary keys and foreign keys are determined by resolving synonyms and homonyms. Then, horizontally partitioned classes are refined into single classes, and associations and generalisations are identified using the evaluation of keys. Finally, OO classes are refined through eliminating redundant associations. Fahrner and Vossen [1995b] described a method in which

an RDB schema is normalised to 3NF, enriched by semantics using data dependencies, and translated into an ODMG-93 ODL schema. This method makes extensive use of inclusion and exclusion dependencies. Moreover, the resulting schema is then restructured by the user with respect to OO paradigm options, e.g., binary relationship relations are eliminated and integrity constraints are mapped into class methods. Castellanos [1993]; Castellanos et al. [1994] presented a method that generates the BLOOM [Abelló et al., 1999] schema from an RDB. The method consists of two phases. An RDB schema is improved semantically based on a knowledge acquisition process to discover implicit semantics by analysing the schema and data instances. Then the enriched RDB schema is converted into a BLOOM schema. The knowledge acquisition phase involves the determination of keys and their types, of data dependencies such as functional, inclusion and exclusion dependencies, and of the normalisation of the schema to 3NF. However, unlike in Premerlani and Blaha [1994] method optimization structures, e.g., horizontal decomposition or different representations of complex attributes are not considered. Fong [1997] suggested a sound theoretical method for converting RDBs data into OODBs. Relation tuples are converted, downloaded into sequential files, and then reloaded into the OODB. However, weak entities and multi-valued and composite attributes are not clearly tackled in this work. Ramanathan and Hodges [1996, 1997] presented a method for mapping an RDB schema that is at least in 2NF into an OODB schema without the explicit use of inclusion dependencies, and without changing the existing schema. All of the information required during the process comes from information on primary keys and foreign keys. However, the method also addresses database optimisation issues such as BLOBs, horizontal and vertical partitioning, which cannot be mapped into object schema without using data dependencies. Zhang et al. [1999] described a method based on MVD to remove data redundancy and update anomalies. A composition process is proposed to reduce the input RDB schema. Then, the simplified relations are mapped into equivalent OO classes.

Clustering RDB relations-to-OODB: Yan and Ling [1993] presented a method that produces an OODB schema from an RDB using a clustering technique, in which clusters of relations that represent object classes, aggregation, association and inheritance relationships are identified. A strong entity is wrapped with all of its direct weak entities, forming a complex cluster which holds the strong entity name. In the

case of deep levels of clustering, a dominant entity may aggregate its component entities if they have no relationships with other entities. The method proposes generating OIDs for identified objects by concatenating the key values of each tuple with the relation name. Missaoui et al. [1995, 1998] adapted the clustering technique proposed by Teorey et al. [1989] to produce a clustered EER diagram. In this method, related entities are identified and defined as one unit. The diagram produced is then translated into an OO schema.

RDB-to-*RID graph*-to-OODB: Alhajj and Polat [2001] re-engineered an RDB into an OODB using the RID graph. The RID graph, which is similar to EER model is derived and optimised in order to identify relationships. Finally, RDB tuples are converted into objects in an OODB.

RDB-*SOT*-OODB: Behm et al. [2000] proposed a model called SOT to facilitate RDBs migration. An RDB schema is mapped into the SOT schema, which is then converted into an OO schema. An SOT schema consists of a set of SOTs where each has a set of attributes of basic type, collection and reference. The references represent the relationship between SOTs. Every SOT and attribute is identified by a unique identifier to avoid naming conflicts. Transformation rules consist of five parts, namely, definitions, patterns, preconditions, schema and data operations [Behm et al., 1997]. The data migration process is accomplished automatically.

Nesting RDB relations-to-OODB: Singh et al. [2004] proposed an algorithm for mapping an RDB schema into a corresponding OO schema based on common attributes factoring. However, neither constraints nor resolving synonym and homonym issues are considered.

3.4 Migrating RDB into ORDB

This section reviews existing proposals for transforming conceptual models into ORDBs. Transforming conceptual models (e.g., EER, UML class diagrams) into ORDB have been studied extensively over the past ten years [Stonebraker et al., 1999; Marcos et al., 2001; Mok and Paper, 2001; Soutou, 2001; Urban et al., 2001; Marcos et al., 2003; Pardede et al., 2004; Arora et al., 2005; Eessaar, 2006; Grant et al., 2006; Mok, 2007]. A common finding from these studies is that the logical structure of an ORDB schema is achieved by creating object-types from UML diagrams. Tables are created

based on the pre-defined object-types. An association relationship is mapped using `ref` or a collection of `ref`s depending on the multiplicity of the association. Multi-valued attributes are defined using arrays/nested tables. Inheritance is defined using foreign keys or `ref` types in Oracle 8_i and the `under` clause in Oracle 9_i/SQL3 [Marcos et al., 2003].

A method of mapping and preserving collection semantics into an ORDB has recently been proposed [Pardede et al., 2004]. The method transforms UML conceptual aggregation and association relationships into ORDB using `row` and `multiset` provided by SQL4 [Pardede et al., 2003]. More recent work has focused on mapping UML aggregation/composition relationships into ORDBs [Marcos et al., 2003; Eessaar, 2006]. Urban et al. [2000, 2001] described essential rules for converting UML class diagrams into ORDB schemas, using triggers to preserve inverse relationships between objects for bi-directional relationships. Marcos et al. [2001, 2003] proposed new UML stereotype extensions for an ORDB design, focusing on aggregation and composition relationships. Urban and Dietrich [2003] presented a method using UML diagrams as a foundation for analysis, transforming them into RDB/OODB/ORDB schemas.

Grant et al. [2006] have compared and evaluated most of the above and others similar proposals. Their analysis might aid in the standardisation of these techniques and the development of a tool that could support in ORDBs design. Although most ORDB concepts are present in these proposals, their focus has been on the design of ORDBs rather than on migration. However, if a migration process uses a conceptual model as an intermediate stage, then these proposals could be useful in schema translation.

3.5 Migrating RDB into XML

Due to a wide adoption of XML for Internet business, the migration of RDBs into XML databases has become more important. This section presents a review of proposals for migrating RDBs into XML. Some work uses data dictionaries and assumes well-designed RDB [Du et al., 2001; Alhajj and Polat, 2001; Lee et al., 2002] whereas others consider legacy RDB for migration into XML documents [Wang et al., 2004, 2005]. Besides, the resulting XML schemas might be a DTD [Lee et al., 2002], XML Schema [Wang et al., 2005] or independent XML language [Dobbie et al., 2000]. However, several researchers have proposed ways to transform UML class diagrams to XML [Conrad et al., 2000; Vela and Marcos, 2003].

RDB-to-*ER*-to-XML: Wang [2004]; Wang et al. [2005] proposed a method focusing on legacy RDBs. Firstly, the ER model is extracted from the RDB by applying the DBRE technique described by Alhajj [2003], which results in an RID graph. Then, the RID graph is mapped into an XML Schema. The structure of the generated XML document is based on user specification into a flat or nested structure. Each entity in an ER model is transformed into an XML complex type. Each attribute is mapped into a sub-element within a related complex type. Relationships among entities are mapped using `key` and `keyref` elements. After the schema is translated, an XML document is generated from RDB data. However, inheritance and aggregation relationships are not considered properly in this study.

RDB-to-*EER*-to-XML: Fong and Cheung [2005] introduced a method in which data semantics are extracted from an RDB schema into an EER model, which is then mapped into an XSD graph. The XSD graph captures relationships and constraints and is mapped in turn into an XML Schema. However, the authors suggested mapping foreign keys into a hierarchy of element/sub-elements, which may cause redundancy when an element has a relationship with more than one element. Fong et al. [2001] described a method for translating an RDB schema into an XML-Data schema [Layman et al., 1998] and then converting the data into an XML document. They capture semantic information from an RDB using an EER model, giving the basis for generating an XML document. However, some relationship types, e.g., M:N, n-ary are not considered. Fong et al. [2006] used the EER model to enrich an existing RDB semantically and translating it into a corresponding DTD schema. The RDB data are converted and loaded into an XML document according to the translated DTD schema.

RDB-to-DTD: Laforest and Boumediene [2003] described two algorithms to extract data-centric and paragraph-centric DTD from RDBs automatically. One table is determined to be the main root element, and then columns of that table, which are neither primary key nor foreign keys are mapped as its sub-elements. The primary key is added to its root elements as an attribute. Other tables that hold the primary key of the root table as foreign keys are translated as sub-elements with cardinality "*". For each foreign key included in the primary key, a new sub-element with PCDATA type is generated, holding the same name as its reference table. Foreign keys that are not included in the primary key are converted into sub-elements in the root. Their composition then has to be defined, and their cardinalities defined as "1" or "?"

depending on integrity constraints.

EER-to-XML: Kleiner and Lipeck [2001] translated an EER model to DTD. However, some data semantics cannot be represented, e.g., the limitations of DTD in specifying composite keys. Moreover, some relationships, i.e., inheritance and aggregation are not considered in this work. The work has been extended considering inheritance relationships, and generate an XML Schema from an EER model [Pigozzo and Quintarelli, 2005]. However, the algorithm tries to create a hierarchal structure that is deeper rather than larger. This may cause redundancy or disconnected elements in the resulting XML document. Liu and Li [2006] devised a set of mapping rules to transform an ER diagram into an XML schema.

UML-to-XML: Conrad et al. [2000] proposed a method for transforming UML into DTD in the context of OO software design. Vela and Marcos [2003] proposed a method for extending UML to represent an XML Schema in graphical notation, which has a unique equivalence with an XML Schema. However, although UML can model data semantics such as aggregation and inheritance, it is still weak and unsuitable in handling the hierarchal structure of the XML data model [Fong and Cheung, 2005]. There is other work in this direction for mapping UML class diagrams into an XML Schema [Routledge et al., 2002; Krumbein and Kudrass, 2003] or a DTD [Kudrass and Krumbein, 2003]

RDB-to-*ORA-SS*-to-XML Schema: Du et al. [2001] developed a method that employs an ORA-SS model to support the translation of an RDB schema into an XML Schema. They proposed a variety of translation rules for converting a semantically enriched RDB schema into an ORA-SS model [Dobbie et al., 2000], which in turn is then translated into the XML Schema. However, they adopted an exceptionally deep clustering technique, which is prone to errors such as data redundancy, loss of semantics and breaking of relationships among objects.

RDB-to-*DOMs*-to-DTD document: Fong et al. [2003] proposed a procedure to translate RDBs into XML documents. Based on data dependency constraints, this work de-normalises an RDB into joined tables, which are then translated to document object models (DOMs). These DOMs are integrated into one DOM, which is then mapped into a DTD schema. Based on the DTD schema generated and data dependencies, each tuple of the joined tables is loaded into an object instance in DOM and then transformed into a DTD document.

NeT and CoT algorithms: Lee et al. [2001] presented the flat translation (FT) algorithm that maps RDB tables into DTD elements. However, the algorithm neither utilises features provided by the XML model nor considers integrity constraints. Another algorithm known as nesting-based translation (NeT) has been proposed to remedy the drawbacks of FT using an iterated mechanism of the `nest` operator to generate nested structures of DTD schema from relational inputs [Lee et al., 2002]. However, this algorithm has some limitations, e.g., the mapping of each table separately and the nesting operations are too time consuming, as all tuples in a table need to be scanned repeatedly to achieve the best nesting outcome. Together with NeT, Lee et al. [2002] presented a constraints-based translation (CoT) algorithm that considers the preservation of integrity constraints.

3.6 Discussion

The investigation into the problem of RDB migration shows that proposals made so far have had different focuses. Each proposal has made certain assumptions to facilitate the migration process, which might be a point of limitations or a drawback. While existing works for migrating into OODBs focus on schema translation using source-to-target techniques, we have noted that most works for migrating to XML have used source-to-conceptual-to-target techniques, focusing on generating a DTD schema and data. Moreover, all research on the generation of ORDBs has focused on design rather than migration. It could be concluded, based on our analysis of the literature, that research into the migration of RDBs into object-based/XML databases is still immature, and that therefore several areas are in need of further attention.

Due to their focus on schema rather than data, the proposals reviewed above either ignore data conversion or assume working on virtual target databases (using mapping and gateways middleware) and data retain stored in RDBs. Moreover, there are still shortcomings in the implementation of RDB data conversion in a more effective manner into more than one environment. Using middleware may lead to slow performance, making the process expensive at run-time because of the dynamic mapping of tuples to complex objects [Behm et al., 2000; Behm, 2001]. However, using object-based DBMSs and native-XML, objects can be stored and retrieved directly without any need for translation layers, hence saving development time and increasing performance.

Some semantics (e.g., inheritance, aggregation) are not considered in some work. This is mainly due to their lack of support for such semantics either in source or target data models, e.g., ER model and DTD lack support for inheritance. Despite the ability of UML to model data semantics such as aggregations and inheritances, UML is still weak and unsuitable for handling the hierarchical structure of the XML data model [Fong and Cheung, 2005]. Although inheritance relationships could be indirectly realised in an RDB, they have been either ignored or only briefly considered. Different types of inheritance have not been tackled, such as unions, mutual exclusion, partition and multiple-inheritance; and neither have their constraints, e.g., total/partial, disjoint and overlapping. Translating inheritance relationships from RDBs to object-based/XML databases and capturing their data semantics, needs more attention.

There has been less effort to use standards such as the ODMG 3.0, SQL4 and XML Schema as target models. The adoption of standards is essential for better semantic preservations, portability and flexibility. In the ODMG 3.0 model, referential integrity is maintained automatically via inverse references. SQL4 has the ability to address complex objects in ORDBs. Compared to DTD, the XML Schema offers a much more extensive set of data types, and provides powerful referencing, nesting and inheritance mechanisms of attributes and elements.

The majority of work so far has generated databases that are either like flat relational, in which object-based model features and the hierarchical form of the XML model are usually missed, or have deep levels of clustering/nesting, which may cause data redundancy. It would be desirable to avoid the flattened form and to reduce the levels of clustering object structures as much as possible in order to increase the utilisations of the target models and to avoid undesirable redundancy. This requires the preservation of the semantics of the source database into a conceptual model, which takes into account the relatively richer data model of the target database environment. The success of the migration process depends on the extent to which data semantics are retained in the conceptual model and how they are translated into a target database.

Although known conceptual models, e.g., ER, EER and UML may be used as intermediate representations during RDB migration, it has been argued here that they are not appropriate for the characteristics and constructs of more than one target data model, and are not supporting data representation. UML should be extended

by adding new stereotypes or other constructs to specify the peculiarities of ORDB and XML models [Marcos et al., 2003; Vela and Marcos, 2003]. In addition, several dependent models have been developed for specific applications, but these are inappropriate to be applied to generate three different data models. The SOT model [Behm et al., 2000] has been designed only for migrating RDBs into OODBs. The BLOOM model [Abelló et al., 1999] was defined for different local schemas to be integrated in federated systems, whereas the ORA-SS model [Du et al., 2001] has been designed to support semi-structured data models.

The evaluation of the different techniques and proposals has shown that very few of the existing studies provide solutions to the problems mentioned above or to the general problem raised in Chapter 1. Viewing objects on top of existing RDBs and establishing gateways to access existing data for only data retrieval purposes cannot solve the problem of mismatch between different paradigms or preserve RDB data semantics. In addition, the existing work on database migration does not provide a complete solution for more than one target database for either schema or data conversion. Besides, none of the existing proposals can be considered as a method for migrating an RDB into an ORDB. An integrated method, which deals with migration from RDB to OODB/ORDB/XML, which covers both schema and data does not yet exist.

3.7 Summary

This chapter has provided a survey of the approaches and techniques used so far for migrating RDBs into other databases, i.e., OODB, ORDB and XML. Three aspects of migration have been discussed: semantics enrichment, schema translation and data conversion. There are three approaches used to accomplish database conversion: 1) viewing objects on top of RDBs where data is processed in object/XML form and stored in relational form; 2) database integration where a gateway is used on top of multiple heterogeneous databases to support a single view; and 3) database migration where an RDB is migrated into its equivalents. On the other hand, translation techniques are divided into two categories: i) source-to-target translation, in which a source database is translated directly into a target database, and ii) source-to-conceptual-to-target translation, in which a source schema is enriched by semantics or recovered to a conceptual schema before being translated into a target schema. The

source-to-target translation includes flat, clustering and nesting translation. The proposals for RDB migration in the literature have been discussed in separate categories according to the different target databases. Within each category, existing proposals have been compared in terms of translation techniques, prerequisites, and specific features. The aims have been to provide a comprehensive view of the problem of RDB migration, to review various techniques and proposals, to identify their commonalities and differences, to assess the impact of pervious research, and to show how it has shaped current and future research in this area.

Based on the investigation into the general problem of RDB migration and the open problems mentioned in this chapter, we propose a complete method called MIGROX, which preserves the structure and semantics of an existing RDB in a CDM to generate OODB, ORDB and XML. Chapter 4 introduces the MIGROX solution, where a detailed description of the solution is then presented in Chapters 5-7.

Chapter 4

An Overview of the Proposed Method

Chapter 3 provided a detailed review of existing RDB migration techniques and proposals. As mentioned in Section 1.3, we have proposed a method called MIGROX for migrating RDBs into object-based and XML databases. The method has three phases: semantic enrichment, schema translation and data conversion. The purpose of this chapter is to provide an overview of MIGROX.

This chapter is organised as follows. Section 4.1 provides an introduction to MIGROX. Section 4.2 reviews the assumptions on which MIGROX is based. Section 4.3 introduces the semantic enrichment phase. Sections 4.4 and 4.5 provide an overview of the schema translation and data conversion phases, respectively. Section 4.6 summarises the chapter and points to what follows.

4.1 Introduction

The migration of RDBs into relatively newer databases, i.e., OODBs, ORDBs and XML has been motivated by the dominance of traditional RDBs in the marketplace and their limitations in supporting complex structures and user-defined data types provided by these new technologies. The problem is how to effectively migrate an existing RDB as a source into the newer databases as targets, and what is the best way to enrich the semantics and constraints of the RDB in order to appropriately capture the characteristics of these targets? We tackle this question by proposing the MIGROX solution for migrating an RDB into these targets based on available

standards. This section outlines the main principles of the solution. The remaining sections briefly describe the different phases of the solution, whereas detailed descriptions are provided in Chapters 5-7.

MIGROX takes an existing RDB as an input, enriches its metadata representation with required semantics, and constructs an enhanced relational schema representation (RSR). Based on the RSR, a canonical data model (CDM) is generated, which captures the essential characteristics of the target data models, for the purpose of migration. Due to the heterogeneity of the three target data models, we believe that it is necessary to develop a CDM to bridge the semantic gap among them and to facilitate the migration process. The CDM is designed to preserve the integrity constraints and data semantics of the RDB so as to fit in with the target database characteristics. In other words, MIGROX preserves the structure and semantics of an existing RDB to generate OODB, ORDB and XML schemas, and effectively converts existing RDB data into target databases without redundancy or loss of data.

The method is more beneficial compared to the existing solutions as it produces three different output databases based on the user choice, as shown in Figure 4.1. In addition, the method exploits the range of powerful features provided by target data models, i.e., ODMG 3.0, SQL4 and XML Schema. MIGROX has three phases: 1) semantic enrichment, 2) schema translation, and 3) data conversion. In the first phase, the CDM is produced which is enriched with the RDB's constraints and data semantics that may not have been explicitly expressed in its metadata. The CDM so obtained is mapped into target schemas in the second phase. We have designed sets of rules which are integrated in algorithms to translate the CDM into each of the three target schemas. The third phase converts RDB data into its equivalents in the new database environment. We have developed algorithms for converting source data into targets based on the CDM.

To demonstrate the effectiveness and validity of MIGROX, a prototype has been developed, which realises the method's concepts and algorithms, and generates target databases successfully. Chapter 8 explains how the prototype has been developed, and Chapter 9 shows how the experimental study was conducted to evaluate the prototype.

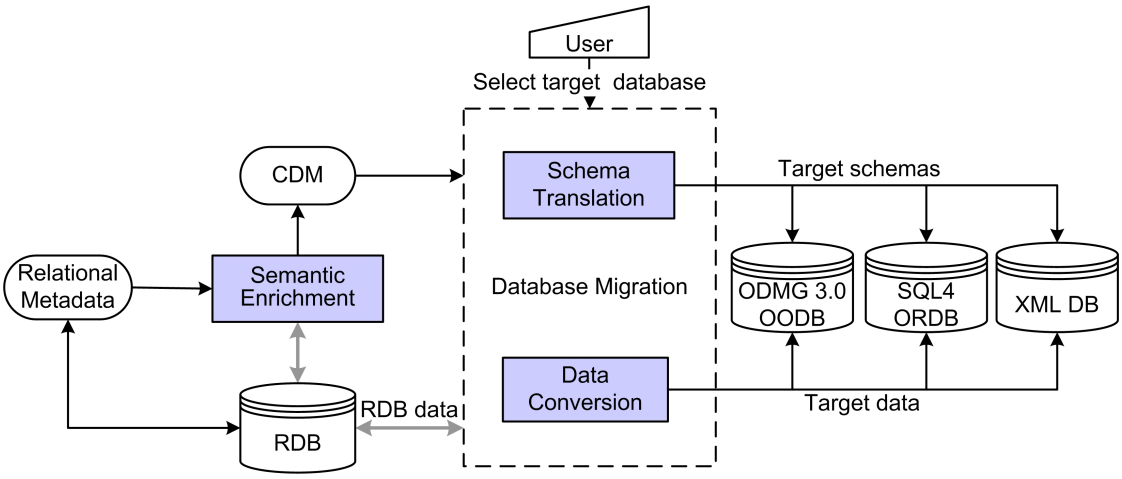

Figure 4.1: Schematic view of MIGROX

4.2 Work Assumptions

MIGROX relies on a number of assumptions as follows:

- To get the best results, it is preferable that the migration process is applied to an RDB that is at least in 3rd normal form (3NF) and which contains base relations rather than views. An RDB schema is in 3NF if every relation is in 2NF and non-primary key attributes are only dependent on the primary key, so that no relation has transitive dependency [Elmasri and Navathe, 2006]. A relation that is not in 3NF may have redundant data, update anomalies or no clear semantics of whether it represents one real world entity or relationship type. These three related problems may affect the real world meaning materialised in object-based/XML models. Besides, the advantages provided by such models, which have motivated migrating RDBs into them, are not exploited. As a result, the target database may be flatter than expected. However, MIGROX can be applied to RDBs in both 1NF and 2NF when essential information is available, e.g., primary keys and foreign keys. Unlike with base relations, views may combine attributes from various entity and relationship types, although RDMSs may materialise repeatedly used views in order to reduce the JOIN operations of base relations for performance reasons [Elmasri and Navathe, 2006].

- Data dependencies are represented by primary keys and foreign keys. For each foreign key value there is an existing, matched primary key value which can

be considered as a value reference. The kinds of RDB relations and relationships are identified using primary/foreign keys, i.e., through keys' composition of each other. Other representations may lead to different target database constructs. For example, a relation L is a strong relation if its primary key is not fully or partially composed of any foreign keys. Similarly, L is a sub-class if its primary key is entirely composed of the primary key of a super-class relation. In addition, in EER models, an inheritance relationship is represented using generalisation/specialisation, which has different types such as total, partial, disjoint and overlapping. However, such types of specialisation can be represented (indirectly) in relational data models in many alternative ways [Elmasri and Navathe, 2006]. The most common alternative represents inheritance as one relation for a super-class and one relation for every sub-class. The super-class is represented by a relation L with its primary key $K(L) = k$ and attributes $Attrs(L) = \{k, a_1,..., a_n\}$. Each sub-class relation S with its attributes is represented by relation $S(k$, attributes of $S)$ and $K(S) = k$. This alternative works for a total, partial, disjoint or overlapping specialisation. Another alternative is to have one relation for each sub-class, containing the attributes and key of the super-class. Here there is a relation S for each sub-class, where $Attrs(S) = \{$attributes of $S\} \cup \{k, a_1,..., a_n\}$ and $K(S) = k$. This alternative works only when each entity in the super-class belongs to at least one of the sub-classes. In addition, an inheritance can be represented using null values in tuples, by examining inclusion dependencies in DDL and queries in DML specifications [Akoka et al., 1999]. However, MIGROX assumes the first alternative because it is based on primary/foreign key matching, and without the user's help it would not be possible to automatically identify the other alternatives.

- Querying the data in databases is used to extract cardinality constraints that cannot be extracted from metadata. Cardinality determination depends on whether or not a foreign key is nullable or/and unique. However, metadata provides only information that a primary key is referenced by a foreign key but it does not provide information as to how many instances of a foreign key point to how many instances of a primary key. In addition, it may not be possible to obtain information about attribute nullabilty from the data dictionary, so that the data content may be examined (analysed or queried). Therefore, we assume that the RDB being migrated is complete. Examining or querying data

instances in order to extract cardinality constraints may not yield the semantics of an existing RDB if the data are incomplete. Different database states give different information about cardinality. For example, assume the `Employee` and `Project` relations participate in an M:N relationship, as each employee is working on several projects and each project is staffed by many employees. However, if data instances show that every project is staffed by many employees, but each employee is working on only one project, then cardinality from the data would indicate a 1:M instead of an M:N relationship.

- The data dictionary is used to provide metadata information. The existence of a modern RDBMS, and hence a database on which it operates, is assumed.

4.3 Semantic Enrichment

Semantic enrichment is a process of analysing and examining a database to capture its structure and definitions at a higher level of meaning. This is achieved by enhancing the representation of an existing database's structure in order to make hidden semantics explicit. The success of the process depends on the amount of information that can be extracted from the existing schema, the method that is followed to extract that information, and the technique by which an enriched semantic representation is constructed. Consequently, additional domain semantics need to be investigated, such as the classification of relations and attributes, and the determination of relationships and cardinalities. In our approach, the semantic enrichment phase involves extraction of data semantics of an RDB by obtaining a copy of its metadata and enriching it with required semantics, and constructing an enhanced RSR. Based on the RSR, a CDM is generated which captures the essential characteristics of target databases (i.e., object-based and XML). In this section, the thinking behind RSR and CDM is presented, including why they are needed, what purpose they serve, and their definitions. The algorithm for constructing the RSR from an input RDB is also explained. An algorithm has been developed for generating a CDM from RSR and an existing RDB data, which is described in detail in Chapter 5.

The main benefit of using RSR and CDM together is that an RDB is read and enriched once whereas the result can be used many times to serve different purposes (e.g., schema translation and data conversion). This facilitates migration into new

target databases without repeatedly referring to the existing RDB. Figure 4.2 shows a schematic view of the semantic enrichment phase. The process starts by extracting the basic metadata information about a given RDB (e.g., relation and attribute names, keys, etc.), leading to the construction of an enriched structure, i.e., RSR, which is designed in such a way so to ease key matching for the classification of RSR's constructs. The next step is to identify CDM constructs based on a classification of the RSR constructs, including relationships and cardinalities, which is performed through data access. Finally, the CDM structure is generated.

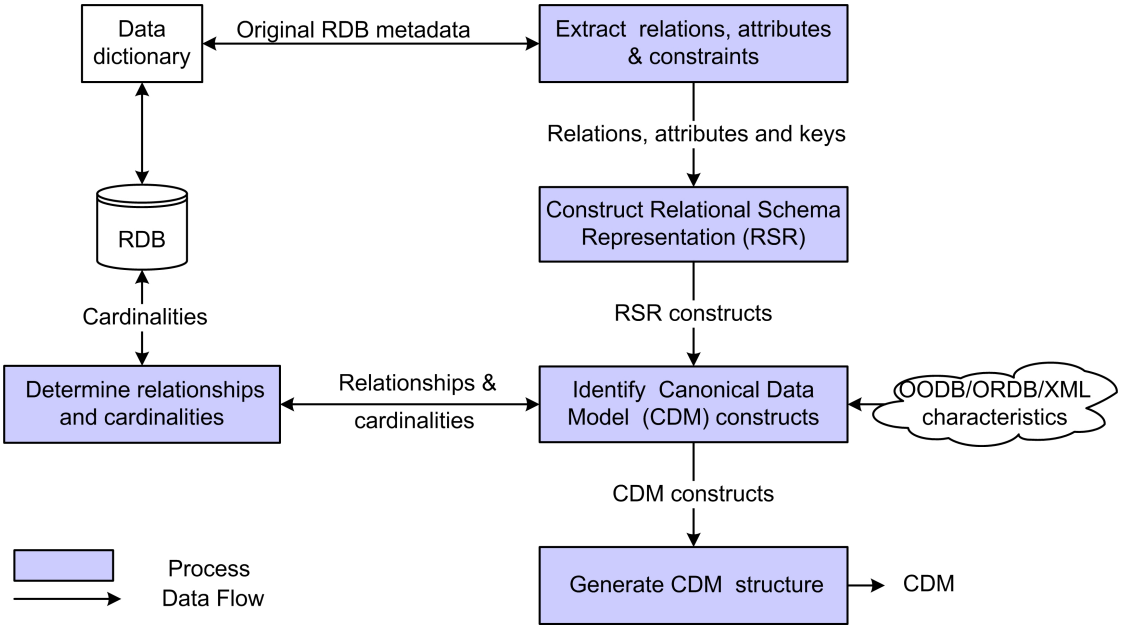

Figure 4.2: Schematic view of the semantic enrichment process

In databases, the essential semantics come with their schemas, whereas further semantics might be hidden in application programs. Data semantics can be extracted using a variety of ways such as from catalogues (i.e., data dictionaries), DBRE tools, specified by users interactively, or from conceptual schemas and design documents. However, conceptual schemas may not be recovered precisely in the DBRE from the final logical or physical schemas, and a full understanding of a database is easily lost when experienced users are unavailable or design documents are missing [Alhajj, 2003]. In RDBMSs, metadata is usually stored in data dictionaries, which can be accessed to derive information about database structures.

4.3.1 Necessary Information

The information that is needed about an RDB schema for conversion into a target schema depends on the degree of automation of the migration process and the kind of semantic information that is needed for the target databases. Relations in RDBs are designed conventionally from entities and relationships in a conceptual schema. Therefore, relations represent entities or relationships among entities. Relations include attributes, primary keys, and foreign keys. Each attribute has a type and arbitrary domain. Binary relationships can be 1:1 or 1:M. Relationship relations are n-ary where n $\geq$2. Information about RDBs can be obtained readily from data dictionaries and data content via SQL queries.

The basic information needed to proceed with the semantic enrichment phase includes relation names and the properties of attributes, including attribute names, data types, length, default values, and whether or not the attribute is nullable. Relations and their attributes need to be classified for systematic mapping into the corresponding structure of the target data model.

The most important information needed for relationship identification concerns data dependency. Functional and inclusion dependencies, which are basically used to enforce data integrity, can be recovered from primary keys, foreign keys and unique keys. Each relation's attribute value functionally depends on a primary key value. A primary key of a relation L may be referenced by another relation L_1 (exported from L to L_1) for relationship participation, where the primary key is then called a foreign key of L_1. At the same time the foreign key is then called an exported key of L; thus the inverse of a foreign key is an exported key. Exported keys play an important role in OODBs/ORDBs, which support bi-directional relationships. Extracting and matching keys is sufficient for the classification of relations, e.g., as strong or weak, the resolution of synonyms and homonyms, and for the identification of relationships, e.g., aggregation or association.

The analysis of schema and data is required to obtain domain semantics such as relationship types, which may involve binary, n-ary, whole/part and super-class/sub-class. Besides, information concerning cardinality constraints and whether a relationship is optional or mandatory is also needed in order to generate the CDM correctly.

Much of the semantics needed for the enrichment process may not be found in an RDB schema due to poor design or limited RDB expressiveness [Chiang et al., 1994].

Some of the semantic information needed for the target schemas cannot be found in an RDB schema, e.g., class methods. An RDB schema might not be in 3NF due to poor database design or for performance reasons. Generally speaking, a relational model has less structural and behavioural expressiveness compared to other data models [Saltor et al., 1991], so that an RDB may not directly represent a user's conceptualisation. In contrast to RDBs that model a static part of entities, object-based models capture more semantics by specifying dynamic object behaviour. Compared to the richer data models, the relational data model provides less structural expressiveness, e.g., it does not support inheritance directly. Moreover, the relational data model provides relatively less behavioural expressiveness because, for example, it does not support the definition of new operations rather than generic operations [Saltor et al., 1991]. Therefore, a portion of the intended semantics would be lost either due to this limited expressiveness [Chiang et al., 1994] or for mapping a non-3NF schema. User interaction might be necessary to provide basic missing semantics.

4.3.2 Relational Schema Representation (RSR)

This section introduces RSR as a representation of an RDB's metadata, which is used as a source of information for the generation of CDM. An RDB schema S can be defined as a set of relations, $S = \{R_1, R_2,..., R_n\}$ and a set of integrity constraints [Elmasri and Navathe, 2006]. Each relation R_i is a finite set of attribute names $\{a_1, a_2,..., a_m\}$. Corresponding to each attribute name a_i is a set of an arbitrary domain. The RDB instance DB of S is a set of relation instances, DB = $\{r_1, r_2,..., r_m\}$. Each r_i is a state of R_i and must satisfy specified integrity constraints. An RDB schema is typically defined and created using SQL DDL statements.

Definition 4.3.1. An RDB schema is denoted in our approach as a set of a 6-tuple: RSR := $\{R_{rsr} \mid R_{rsr} := \langle r_n, A_{rsr}, PK, FK, EK, UK\rangle\}$, where:

- r_n denotes the name of a relation R_{rsr}.
- A_{rsr} denotes a set of attributes of R_{rsr}: $A_{rsr} := \{a \mid a := \langle a_n, t, l, n, d\rangle\}$, where a_n is the name of attribute a, t its type, l its data length, n whether nullable or not ('y'|'n') and d a default value if given.
- PK denotes the primary key of R_{rsr}: $PK := \{\alpha \mid \alpha := \langle pa, s\rangle\}$, where α represents one key attribute, pa is an attribute name and s is a sequence number

in the case of a composite key; however, s is assigned as 1 in the case of a single valued key.

- FK denotes a set of foreign key(s) of R_{rsr}: $FK := \{\beta \mid \beta := \langle er, \{\langle fa, s\rangle\}\rangle\}$, where β represents one key (whether single or composite), er is the name of an exporting (i.e., referenced) relation that contains the referenced primary key, fa is an attribute name, and s is a sequence number.
- EK is a set of exported key(s) of R_{rsr}: $EK := \{\gamma \mid \gamma := \langle ir, \{\langle ea, s\rangle\}\rangle\}$, where γ represents one key, ir is the name of an importing (i.e., referencing) relation that contains the exported attribute name ea (i.e., foreign key attribute).
- UK is a set of unique key(s) of R_{rsr}: $UK := \{\delta \mid \delta := \{\langle ua, s\rangle\}\}$, where δ represents one key, ua is an attribute name and s is a sequence number.

The RSR provides an image of the metadata in primary memory obtained from an existing RDB's secondary storage. The efficient construction of RSR overcomes the complications that occur during matching keys in order to classify relations (e.g., strong or weak), attributes (e.g., non-key attribute) and relationships (e.g., M:N, inheritance, etc.). The relation R_{rsr} is constructed with its semantics as one 6-tuple, which is easily identifiable and upon which set theoretical operations can be applied for matching keys. Each part of R_{rsr} describes a specific aspect of it (e.g., A_{rsr} describes its attributes) and all of the parts together describe the structure of R_{rsr}. An important advantage of RSR is that it identifies the set EK, therefore, adding more semantics to an RDB's metadata. The set EK holds keys that are exported from R_{rsr} to other relations as foreign keys.

4.3.3 Algorithm for Extracting RSR

This section presents the **ConstructRSR** algorithm, shown in Figure 4.3, which is used to facilitate the automatic construction of RSR, which contains information extracted from an RDB. The input to the algorithm is an existing RDB metadata and the output is the RSR structure as described in Section 4.3.2. For a given RDB, the algorithm finds the names, attributes and integrity constraints of all the relations, and constructs the RSR. The main steps of the algorithm are as follows:

- For each relation name r in RDB metadata, an RSR relation R (of type R_{rsr}) is defined and given the name of r (i.e., $R.r_n := r$).

```
1: algorithm ConstructRSR (MD: RDB metadata) return RSR
2:    rsr: RSR := ∅ // a set to store RSR relations
3:    foreach relation name r ∈ MD do
4:       R: R_rsr // define an RSR relation as ⟨r_n, A_rsr, PK, FK, EK, UK⟩
5:       R.r_n := r
6:       R.A_rsr := extractAttributes(r)
7:       R.PK := extractPrimaryKey(r)
8:       R.FK := extractForeignKeys(r)
9:       R.EK := extractExportedKeys(r)
10:      R.UK := extractUniqueKeys(r)
11:      rsr := rsr ∪ {R} // add the relation to RSR
12:   end for
13:   return rsr
14: end algorithm
```

Figure 4.3: The **ConstructRSR** Algorithm

- Attributes of r are extracted by calling the *extractAttributes*(r) function, and the result is placed in the set A_{rsr} of R. Each element in the A_{rsr} contains one attribute and its properties.

- The primary key of r is inferred by calling the *extractPrimaryKey*(r) function. Each key attribute *pa* is inferred and given a sequence number s and added to the set PK in pairs $\langle pa, s\rangle$. The sequence s distinguishes between single and composite keys.

- The *extractForeignKeys*(r) function returns the set FK containing the foreign keys of r. Each key attribute fa is assigned s based on matching the key constraint name *con*. Attributes that have the same *con* are given an ascending order of s and together form one key, which is added to the set FK.

- Following the same technique of foreign keys, the *extractExportedKeys*(r) function returns the set EK and the *extractUniqueKeys*(r) function returns the set UK.

- This enables one RSR relation to be constructed with its parts and added as one tuple to the set rsr with a loop back to construct the next tuple. Finally, the algorithm returns the constructed rsr, which is later used together with data stored in an RDB to generate the CDM.

Example 4.3.1. Consider the RDB shown in Figure 4.4. Primary keys are in italics and foreign keys are marked by "*". Table 4.1 gives the RSR constructed for the RDB,

showing only the relations: `Emp`, `Salaried_emp`, `Dept` and `Works_on`. The information includes all attributes and keys for each relation.

EMP

ENAME	*ENO*	BDATE	ADDRESS	SPRENO*	DNO*
Smith	12345	09-Jan-55	31 Hampstead Rd, NE4 8AB	86655	3
Wong	34455	08-Dec-45	16 Hampstead Rd, NE1 7RU	86655	3
Scott	53453	31-Jul-62	4 Northcote St., NE5 5AL	34455	1
Ali	68844	15-Sep-52	49 Hill Street, RG1 2NU	34455	1
Borg	86655	10-Nov-27	162 London Road, OL1 4BX	null	2
Wallace	54321	20-Jun-31	91 St James Gate, NE1 4BB	86655	2

SALARIED_EMP

*ENO**	SALARY
86655	55000
54321	43000

HOURLY_EMP

*ENO**	PAY_SCALE
12345	3
34455	4
53453	2
68844	3

WORKS_ON

*ENO**	*PNO**
12345	1
34534	1
34455	1
12345	2
34534	2
34455	2
68844	3
34455	3
54321	4
54321	5
86655	5

PROJ

PNAME	*PNUM*	PLOCATION	DNUM*
Way Station 1	1	Newcastle	3
Way Station 2	2	London	3
Way Station 3	3	Reading	3
Salford House	4	Salford	2
4 Times Square	5	Reading	1

DEPT_LOCATIONS

*DNO**	*LOCATION*
1	Reading
2	Salford
2	Newcastle
3	London

KIDS

*ENO**	*KNAME*	SEX
12345	Alice	F
12345	Michael	M
34455	Alice	F
34455	Joy	F
54321	Scott	M

DEPT

DNAME	*DNO*	MGR*	STARTD
Accounts	1	86655	19-Jun-71
Administration	2	54321	01-Jan-85
Finance	3	12345	06-Oct-05

Figure 4.4: Sample input RBD

4.3.4 Canonical Data Model (CDM) Definition

This section gives a formal definition of the CDM. The CDM is a source of valuable semantics giving an enriched and well organised data model, which can be converted flexibly into any of the target databases. Besides taking into account the characteristics of the target models, the CDM retains all data semantics that could be extracted from an RDB and the integrity constraints imposed on it. Moreover, it acts as a key mediator for converting existing RDB data into target databases based on the structure and the concepts of the target models. The CDM facilitates the reallocation of attribute values in an RDB to the appropriate values in a target database. Based on the CDM definition, target attributes that represent relationships among classes are materialised into references or changed into other domains.

r_n	A_{rsr}					PK		FK			EK			UK	
	a_n	t	l	n	d	pa	s	er	fa	s	ir	ea	s	ua	s
Emp	eno	int	25	n		eno	1	Emp	spreno	1	Salaried_emp	eno	1		
	ename	char	40	n				Dept	dno	1	Hourly_emp	eno	1		
	bdate	date		y							Works_on	eno	1		
	address	char	40	y							Dept	mgr	1		
	spreno	int	25	y							Kids	eno	1		
	dno	int		n							Emp	spreno	1		
Salaried_emp	eno	int	25	n		eno	1	Emp	eno	1					
	salary	int		y											
Dept	dno	int		n		dno	1	Emp	mgr	1	Emp	dno	1	mgr	1
	dname	char	40	n							Proj	dnum	1		
	mgr	int	25	n							Dept_locations	dno	1		
	startd	date		y											
Works_on	eno	int	25	n		eno	1	Emp	eno	1					
	pno	int		n		pno	2	Proj	pnum	1					

Table 4.1: Results of RSR construction

Canonical models used for database integration should have semantics at least equal to any of the local schemas to be integrated [Saltor et al., 1991]. However, the CDM described here has been designed to upgrade the level of semantics of RDBs, and to play the role of an intermediate stage for migrating RDBs during the schema translation and data conversion phases. It represents many explicit as well as implicit semantics of an existing RDB. Explicit semantics include relation and attribute names, keys, etc.; implicit semantics include the classification of classes and attributes, and relationship names, types, cardinalities, and inverse relationships. Its constructs are classified to facilitate migration into complex target objects, avoiding both flat one-to-one and complicated nesting conversions. Through the CDM, well-structured target databases can be obtained without the proliferation of references and redundancy. However, the richness of CDM may not be fully exploited due to the relatively limited expressiveness of the input RDB. Consequently, some concepts provided by target databases receive less attention in the CDM. For instance, object-based models encapsulate static (i.e., attributes and relationships) and dynamic (i.e., methods) aspects of objects. However, dynamic aspects get less attention here compared to static aspects because an RDB does not support methods and functions attached to tables. Static aspects involve the definition of a class, and its attributes and relationships.

The CDM has three static concepts: class, attribute and relationship. Attributes define class structure, whereas relationships define a set of relationship types. CDM classes are connected through relationships. The CDM can be seen as an independent model, which embraces OO model concepts with rich semantics that can also cater for object-relational and XML data models. In order to express as much semantics as possible, the model takes into consideration some of the features provided by object-based and XML databases, such as association, aggregation and inheritance. The model provides non-OODB key concepts (i.e., foreign keys, null and unique keys)

and explicitly specifies whether attributes and cardinalities are optional or required. Relationships are defined in the CDM in such a way as to facilitate extraction and transformation of data in the data conversion phase, including defining and linking objects using user-defined identifiers. However, the CDM is independent of the RDB from which it has taken the semantics, as well as of any of the target databases to which it could be converted. Real world entities, multi-valued and composite attributes, and relationship relations are all represented as classes in the CDM.

Definition 4.3.2. The CDM is defined as a set of a 6-tuple, representing classes: CDM := $\{Class_{cdm} \mid Class_{cdm} := \langle cn, cls, abs, A_{cdm}, REL, UK\rangle\}$, where each class $Class_{cdm}$ has a name cn, and is given a classification cls and whether or not it is abstract abs. $Class_{cdm}$ has a set of attributes A_{cdm}, a set of relationships REL, and a set of unique keys UK. These properties are described below:

Classification (cls): Classification divides classes into three categories: main classes, component classes and relationship classes. An instance class C_{cdm} of type $Class_{cdm}$ is classified into one of nine different kinds of classes within these categories, which facilitate the translation of C_{cdm} into target schemas:

1. Main classes (classes forming base types in the target database)
    - Regular strong class (RST): a class whose primary key is not composed of any foreign keys.
    - Secondary strong class (SST): an RST class that is inherited by other classes.
    - Sub-class (SUB): a class that inherits another super-class, but is not inherited by other sub-classes.
    - Secondary sub-class (SSC): a sub-class that is inherited by other sub-classes.
    - Secondary relationship class (SRC): a referenced RRC class, an M:N relationship class with attributes, or n-ary relationships where n>2.
    - Regular component class (RCC): a weak class that participates in a relationship with other classes rather than its parent class.
2. Component classes (classes representing multi-valued/composite attributes)

- Multi-valued attribute class (MAC): a class that represents a multi-valued attribute.
- Composite attribute class (CAC): a class that represents a composite attribute.

3. Relationship class (a class describing an M:N relationship between two classes)
   - Regular relationship class (RRC): an M:N relationship class without attributes.

Abstraction (abs): It is important for each super-class to be checked to determine whether it is concrete or abstract class. A super-class is abstract (i.e., $abs :=$ `true`) when all of its objects are members of its sub-classes. Since an abstract class is not instantiable, any data corresponding to an abstract class is subsumed into instances of its sub-classes. A class is not abstract (i.e., $abs :=$ `false`) when all (or some) of its objects are not members of its sub-classes.

Attributes (A_{cdm}): $Class_{cdm}$ has a set of attributes A_{cdm} of primitive data types. $A_{cdm} := \{a \mid a := \langle a_n, t, tag, l, n, d\rangle\}$, where each attribute a has a name a_n, data type t and a tag, which classifies a as a non-key 'NK', 'PK', 'FK' or both primary and foreign key 'PF' attribute. Each a can have a length l and may have a default d value, whereas n indicates whether or not a is nullable ('y'|'n').

Relationships (REL): Classes are characterised through attributes and related to each other through relationships. $Class_{cdm}$ has a set of relationships REL. Each relationship $Rel \in REL$ between a class C_{cdm} and another class C'_{cdm} (of type $Class_{cdm}$) is defined in C_{cdm} to represent an association, aggregation or inheritance.
$REL := \{Rel \mid Rel := \langle relType, dirC, dirAs, c, invAs\rangle\}$, where:

- $relType$ is a relationship type.
- $dirC$ is the name of C'_{cdm}.
- $dirAs$ denotes a set containing the attribute names representing the relationship from C'_{cdm} side.

- *invAs* denotes a set of attribute names representing the inverse relationship from C_{cdm} side.
- c is the cardinality constraint of *Rel* in C_{cdm}.

A relationship is basically a link among objects. The links can have specific cardinalities (occurrence), e.g., one-to-one (1:1) and one-to-many (1:M). A relationship can be an association, aggregation or inheritance. Thus, *relType* can have the following values: '*associated with*' for association, '*aggregates*' for aggregation, and '*inherits*' or '*inherited by*' for inheritance. Relationships in CDM have two cases: 1:M and 1:1. The 1:M relationship (also referred to as M:1) is a common relationship. Two classes can participate in this relationship and the class that holds the primary key would have a set-value of instances of the class that holds the foreign key. In other words, the 1:M relationship is defined by a relationship that represents reference instances of the dominant class (that hold primary key) and a set-value of the weak class (that hold foreign key) instances. The 1:1 relationship is a special case of the 1:M relationship. Therefore, it is similar, except that a single instance replaces a set of instances. The cardinality c is defined by $min..max$ notation to indicate the occurrence of C'_{cdm} object(s) within C_{cdm} objects, where min is a minimum cardinality and max is a maximum cardinality. Based on c, object(s) of C'_{cdm} can be single-valued ($c := 0..1$ for optional relationship or $c := 1..1$ for required relationship), or set-valued ($c := 0..*$ for optional relationship or $c := 1..*$ for required relationship). An n-ary relationship relation, n $\geqslant$ 2, is very difficult to model in an RDB because of normalisation problems, so that an additional intersection relation should be created to hold the primary keys of the relations involved in the relationship. Therefore, the primary key of such a relation is fully or partially formed by the concatenation of the primary keys of two or more relations as disjoint foreign keys. Such relations are represented as two classes in CDM, i.e., RRC and SRC.

- **Association**. An association relationship is a reference from one object to another. In the CDM, an association can exist among non-component classes where $relType :=$ '*associated with*'. Each class may participate in an association with none, one or a set of classes. The relationship is defined bi-directionally between both of the classes participating in the relationship.
- **Inheritance**. An inheritance is a 1:1 relationship, which allows a class (subclass) to inherit the properties of another class (super-class) in addition to its

own properties. The main idea in inheritance is that a super-class could be specialised into one or more sub-classes. Conversely, a sub-class could inherit from one or more super-classes, i.e., multiple-inheritance. However, the CDM does not support multiple-inheritance as target standards, that is, ODMG 3.0, SQL4 and XML Schema do not allow a concrete sub-class to have more than one concrete super-class. Hence, a sub-class inherits only from one super-class. A super-class could be specialised in one or more sub-classes, whereas a sub-class could be generalised by only one super-class. An inheritance can exist among main CDM classes, where $relType :=$ (‘*inherits*’ | ‘*inherited by*’). A sub-class *inherits* its super-class and the latter is then *inherited by* its sub-class(s).

- **Aggregation**. An entity that depends on another entity is called a weak entity, and it has to be identified based on an identifier dependency relationship. This represents an aggregation relationship that formalises a case of complex objects (the composition of one object from other objects), and therefore it allows a class to aggregate one or more other components. In UML, an aggregation is either composition (exclusive), which is conceptually represented by a black-filled diamond, or simple (shared), which is represented by a white-filled diamond [OMG, 2009]. In the composition, the component class (the part) is physically integrated in only one main class (the whole). This means that the life of the part depends on the life of the whole, to which it belongs. This type of relationship is known as aggregation in CDM, which can exist between a main class, e.g., RST class and a component class, i.e., MAC and CAC classes where the $relType :=$ ‘*aggregates*’. A main class can aggregate none, one or many component classes, based on the cardinality c of the relationship. Unlike association and inheritance, an aggregation relationship in the CDM is uni-directional from the whole class into the part class. However, a shared aggregation is a special case of association relationship in UML, and the difference between them is conceptual, i.e., in the UML diagram notation. A part class in the shared aggregation can be shared by more than one whole class. This kind of relationship links two main classes in CDM, and is treated as a normal association.

Unique keys (UK): $Class_{cdm}$ may have a set of unique key(s) that are preserved in UK: $UK := \{\delta \mid \delta := \{\langle ua, s\rangle\}\}$, where δ represents one key, ua is an attribute name, and s is a sequence number.

4.3.5 Algorithm for Generation of CDM

To generate the CDM, we have developed an algorithm called **GenerateCDM**. Given an RSR and RDB data as input, the algorithm generates the equivalent CDM. Using key matching, RSR relations and their attributes are classified, relationships among relations are identified and their cardinalities determined, followed by translation into their equivalents in the CDM. The abstraction of each class in the CDM is then checked. The output of the algorithm, the CDM, is translatable into any of the target schemas and facilitates the conversion of existing RDB data into target databases. The **GenerateCDM** algorithm is described in detail in Chapter 5.

Example 4.3.2. Given the RSR shown in Table 4.1, Figure 4.5 shows the resulting CDM for the `Emp` class in an easy to follow format hiding the deeply nested structure. The `Emp` class, which is an abstract class, classified as SST, has the attributes: *ename*, *eno*, *bdate*, *address*, *spreno* and *dno* with their tags shown. Other properties (e.g., attributes' types, default values) are not shown as these details do not change from RSR to CDM. The class `Emp` is '*associated with*' the classes: `Dept`, `Works_on`, and with itself. Moreover, it '*aggregates*' the `Kids` class and is '*inherited by*' the `Salaried_emp` and `Hourly_emp` classes. Cardinalities are shown for each relationship. Relationships defined here in the form: $relType\ \{invAs \leftrightarrow dirC(dirAs)_c\}$ ($\leftrightarrow$ indicates bi-directional association and $\leftarrow$ indicates uni-directional aggregation). Full descriptions about these classes can be found in Table 5.1.

```
Emp cls:=SST; abs:=true [
   A_cdm := {ename 'NK', eno 'PK', bdate 'NK', address 'NK', spreno 'FK', dno 'FK'},
   REL := {
   associated with {dno ↔ Dept(dno)_1..1, eno ↔ Dept(mgr)_0..1, spreno ↔ Emp(eno)_1..1,
   eno ↔ Emp(spreno)_0..*, eno ↔ Works_on(eno)_1..*},
   aggregates {eno ← Kids(eno)_0..*},
   inherited by {Salaried_emp, Hourly_emp}
         }
    ]
```

Figure 4.5: Sample CDM class schema

4.4 Schema Translation

This section provides an overview of schema translation, the second phase of MIGROX. Schema translation aims to translate the CDM, produced by the semantic enrichment phase into the target schemas as shown in Figure 4.6. The target schemas hold equivalent semantics to that of the existing RDB, which are enhanced and preserved in the CDM. Section 4.4.1 defines the three target schemas, which satisfy the

standards of ODMG 3.0, SQL4 and XML Schema, before introducing the schema translation process in Section 4.4.2.

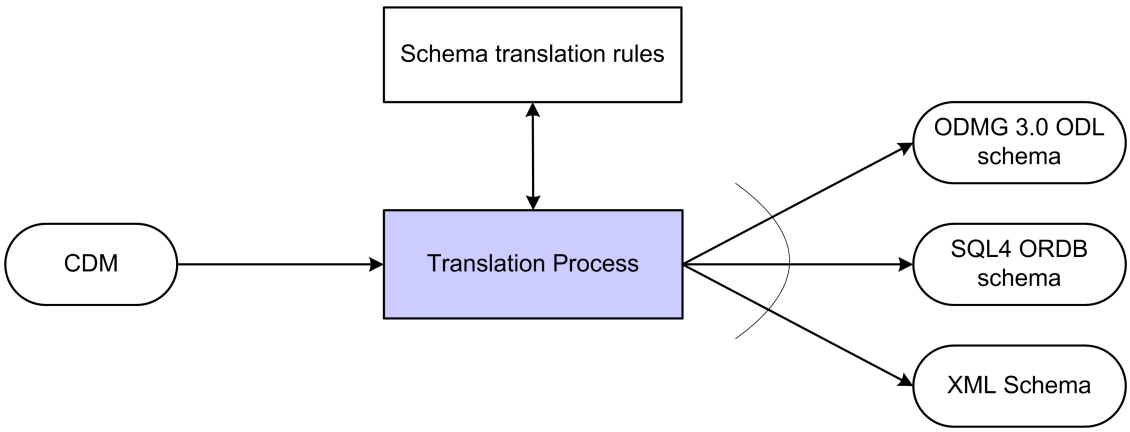

Figure 4.6: Schematic view of translating CDM into target schemas

4.4.1 Target Schemas

This section defines the output target models for schema translation. The translation of the data models defined here into the actual schema definition languages is straightforward.

ODMG 3.0 schema: The potential ODMG 3.0 ODL schema is defined as a set of classes. Each class is identified by a name and may consist of a set of attributes and a set of relationships. The data type of attributes is literal, whereas relationships define associations. Literal types can be (a collection of) primitive/structured attributes. The `set` is the most commonly used unordered collection that allow duplicate, so that it is used to represent literal collection types, and the M side of the 1:M and M:N relationships. Classes can be defined into hierarchies, realising inheritance relationships.

Definition 4.4.1. A target ODMG 3.0 schema is defined as a set of classes:
OOschema := $\{Class_{oo} \mid Class_{oo} := \langle c_n, spr, k, A_{oo}, REL_{oo} \rangle\}$, where c_n is the name of a class $Class_{oo}$, spr is the name of its super-class, k is its key, A_{oo} is a set of its attributes, and REL_{oo} is a set of relationship types in which $Class_{oo}$ participates. The sets A_{oo} and REL_{oo} are defined as follows:

- $A_{oo} := \{a_{oo} \mid a_{oo} := \langle a_n, t, m \rangle\}$, where a_n is the name of an attribute a_{oo}, t is its data type, and m denotes whether a_{oo} is single-valued ('sv') or collection-valued ('cv').

- $REL_{oo} := \{Rel_{oo} \mid Rel_{oo} := \langle rel_n, dirC_n, m, invRel_n \rangle\}$, where rel_n is the name of the relationship Rel_{oo}, $dirC_n$ is the name of the referenced class, m indicates the multiplicity of Rel_{oo}, and $invRel_n$ is the name of the inverse relationship.

SQL4 ORDB schema: The SQL4 ORDB schema is defined as a set of user-defined types (UDTs), and a set of typed tables created based on these UDTs for storing data. Each UDT consists of a set of attributes defined as literal or reference types. Literal types are defined as a primitive, collection of primitive, `row`, or collection of `row` types. Moreover, an attribute can be defined as a reference (`ref`) or a collection of references (`refs`), pointing to a specific UDT. Composite attributes are defined using `row` types. An association relationship is expressed among UDTs using `refs`. The collection (i.e., `set`) of primitive and `row` types are used to define simple and composite multi-valued attributes, respectively, whereas the M side of associations are defined as a collection of `refs`. UDTs and typed tables can be defined into hierarchies, realising inheritance relationships, in which types/tables can be defined as sub-types/sub-tables `under` their super-types/super-tables.

Definition 4.4.2. The SQL4 ORDB schema is denoted as a 3-tuple:
ORschema := $\langle UT, TT, UK_{or} \rangle$, where UT is a set of UDTs, TT is a set of typed tables, and UK_{or} is a set of unique keys. The sets UT and TT are defined as follows:

- $UT := \{udType \mid udType := \langle ut_n, s_{ut}, A_{ut} \rangle\}$, where ut_n is the name of a user-defined type $udType$, s_{ut} is the super-type name of $udType$, and A_{ut} is a set of $udType$'s attributes of literal or ref-based data type:

  $A_{ut} := \{a_{ut} \mid a_{ut} := \langle a_n, t, m, n, d \rangle\}$, where a_n is the name of an attribute a_{ut}, t is its data type, which can be primitive (e.g., integer), user-defined constructed (e.g., `row` type) or ref-based (e.g., `ref`($udType$)); m denotes whether a_{ut} is single-valued or collection-valued, d is a default value in the case of primitive attributes, and n denotes whether or not a_{ut} accepts nulls.

- $TT := \{tTable \mid tTable := \langle tt_n, ut_n, s_{tt}, pk, uoid \rangle\}$, where tt_n is the name of a typed table $tTable$, ut_n is the name of $udType$ based upon which $tTable$ is defined, s_{tt} is the name of its super-table, pk is the primary key of $tTable$, and $uoid$ is the user-defined identifer of the objects of $tTable$.

XML Schema: The structure of an XML document is usually made up of essential components such as annotations, element declarations and type definitions. Moreover, the document may contain other components such as attribute and model groups [Valentine et al., 2002]. Besides, the XML Schema language standard provides declaration of identity-constraints, by which association relationships and integrity constraints can be defined among related elements. Primary key, foreign key and unique key constraints can be defined within the schema's root element using the `key`, `refkey` and `unique` elements, respectively. The `key` and `refkey` elements define relationships, whereas the `unique` element specifies that elements or attributes that are not primary keys must be unique within a specified scope. Moreover, there are several mechanisms in the XML Schema that handle inheritance relationships such as derived types, substitution groups and abstract type mechanisms. Complex types can be derived from other types using the `extension`, `restriction` and `choice` keywords, through which a sub-class complex type extends its super-class type complex `base`. Besides, a super-class complex type can be declared as `abstract` if all its instances are inherited by instances of its sub-classes. The multiplicity of elements is specified by `minOccurs` and `maxOccurs`.

From this, the potential target XML Schema can be generated as two components. One is a global element, which represents the root of the XML tree defined as a complex type, containing schema elements and constraints. The second component is a set, containing all global complex types. Each complex type can be used as a type of one element (or more) declared in the root or in other complex types. As the concept of complex type extension is very similar to object-based single inheritance, we suggest that inheritance is represented using the `complexContent`, `extension` and `base` keywords.

Definition 4.4.3. A target XML Schema is denoted as a 2-tuple:
XMLschema := $\langle Root, GT \rangle$, where *Root* is a global element declared under the schema with its direct local elements and constraints, representing the XML document tree, and GT is a set consisting of global complex types. The types in GT are defined as types of the elements declared in *Root*, or to be referenced by other complex types in GT. The *Root* and the set GT are defined as follows:

- $Root := \langle root_n, LE, PK_x, FK_x, UK_x \rangle$, where *Root* has a name $root_n$, a set of elements LE, and three sets of identity-constraints PK_x, FK_x and UK_x.
  - LE represents the complex type of *Root* that involves a set of local sub-element declarations: $LE := \{e \mid e := \langle e_n, e_t, nim, max \rangle\}$, where each

element e has a name e_n, a type e_t, and a minimum min and maximum max occurrences. The e_t is defined globally under the schema, i.e., in the set GT.

- PK_x is a set of primary keys for the elements defined in the *Root*, where $PK_x := \{pk \mid pk := \langle pk_n, selector, PKfield\rangle\}$. Each key pk has a name pk_n, an element set $selector$ as a scope within which the key is defined, and a set of related sub-elements $PKfield$ selected to be unique.
- FK_x is a set of foreign keys, where $FK := \{fk \mid fk := \langle fk_n, ref, selector, FKfield\rangle\}$. Each foreign key fk has a name fk_n, an element set scope $selector$, a reference constraint name $refer$ that points to a matched primary key name, and a set of related sub-elements $FKfield$.
- UK_x is a set of unique keys, where $UK := \{uk \mid uk := \langle uk_n, selector, UKfield\rangle\}$. Each unique key uk has a name uk_n, an element set scope $selector$, and a set of related sub-elements $UKfield$ selected to be unique.

- $GT := \{compType \mid compType := \langle ct_n, base, abst, LE\rangle\}$, where ct_n is the name of a complex type $compType$, $base$ is the name of its super-type (if it is derived from another type), $abst$ denotes whether or not $compType$ is abstract type, and LE is a set of elements that are declared locally within $compType$.

  $LE := \{e \mid e := \langle e_n, e_t, nim, max\rangle\}$ is defined as for *Root*; however, e_t can be a built-in data type (e.g., a string) or a complex type pre-defined in the set GT.

4.4.2 Algorithms for Schema Translation

When the CDM has been obtained, the schema translation phase is started after the user chooses which target schema is to be produced. Then, an appropriate set of rules is used to map the CDM constructs into equivalents in the target schema. Three sets of translation rules have been designed for mapping CDM into the target schemas. Each rule maps a specific construct, e.g., attribute or relationship. Algorithms have been developed for producing each of the target schemas according to these rules.

The classification of CDM constructs facilitates the identification of their equivalent in the target schema definition languages. Based on cls, each main and concrete CDM class C_{cdm}, where $C_{cdm}.cls \neq$ ('MAC' | 'CAC' | 'RRC'), is translated into a target type. Each type is defined under its super-class if $C_{cdm}.cls :=$ ('SUB' | 'SSC'). However, for component classes, each MAC class is mapped into a multi-valued attribute, and each CAC class is mapped into a composite (e.g., `struct`) attribute. The RRC classes are mapped into an M:N relationship, in which a pair of 1:M relationship is

defined in each of the target types that participate in the relationship. Attributes $C_{cdm}.A_{cdm}$ are translated into equivalents with the same names as in the CDM and their types are converted according to target data types. Primary keys are specified when attributes are tagged with 'PK'. The types of target relationships and their multiplicity are determined by the classification of the CDM class C'_{cdm} related to C_{cdm} being translated, and the properties of each relationship rel defined in C_{cdm}, i.e., $rel \in C_{cdm}.REL$. Each rel is translated into an equivalent target association, aggregation or inheritance relationship. Target relationship names are generated by concatenating $rel.dirC$ with attribute names in $rel.dirAs$, and $C_{cdm}.cn$ with attribute names in $rel.invAs$. The relationship cardinality $rel.c$ is mapped into a single-value when $rel.c := (0..1 \mid 1..1)$ or a collection-value otherwise. The schema translation algorithms and the output schemas are described in detail in Chapter 6.

Example 4.4.1. The ODMG 3.0 ODL schema corresponding to the CDM in Figure 4.5 is shown in Figure 4.7.

```
class Emp (extent Emps key eno) {
   attribute string ename; attribute number eno;
   attribute date bdate; attribute string address;
   attribute set<struct kids{string kname; string sex;}> hasKids;
   relationship Dept manages inverse Dept::manager;
   relationship set<Emp> supervises inverse Emp::supervisor;
   relationship Dept dept inverse Dept::employees;
   relationship Emp supervisor inverse Emp::supervises;
   relationship set<Proj> projects inverse Proj::employees;};
```

Figure 4.7: Sample OODB class schema

4.5 Data Conversion

This section introduces data conversion, the last phase of MIGROX. Traditionally databases are designed based on conceptual models from which the database schemas are derived. Data are then loaded into a database based on the schema created. Semantic relationships among data are modeled using foreign key data values in relational data models. However, in object-based models, the values of attributes are organised into objects, and relationships are mapped directly as inheritance or references using OIDs [Zhang and Fong, 2000]. In MIGROX, in addition to enrichment, the CDM guides the conversion of RDB data into any of the target databases, ensuring that the conversion process is accomplished with data integrity and consistency.

4.5.1 Algorithms for Data Conversion

The data conversion phase concerns the conversion of existing RDB data into the format defined by the target schema. Data stored as tuples in an RDB are converted into complex objects/literals in object-based databases or as element instances in XML documents. The process is performed in three steps as shown in Figure 4.8. Firstly, tuples of each and every relation in the RDB are extracted. Secondly, these data (i.e., tuples) are transformed (converted) to match the target format. Finally, the transformed data are loaded into text files suitable for bulk loading in order to populate the schema generated earlier during the schema translation phase.

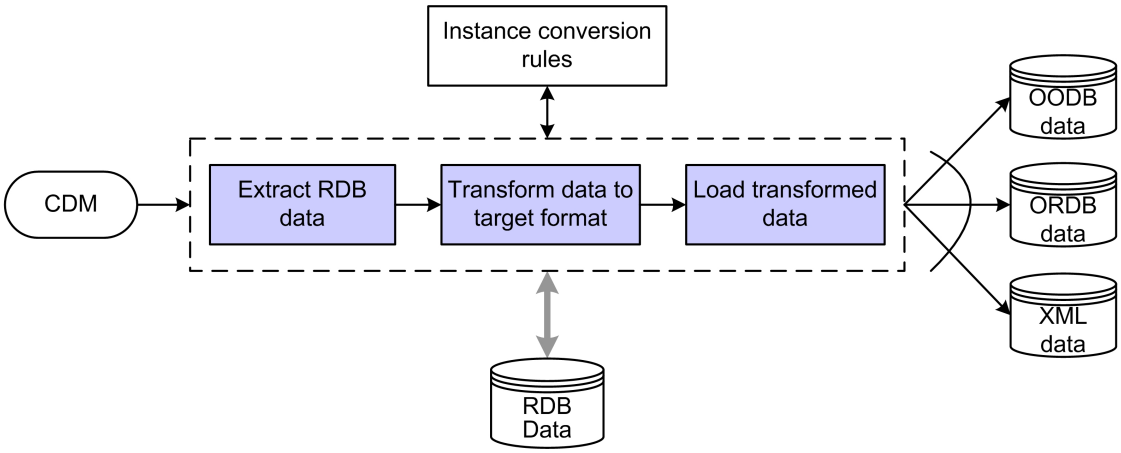

Figure 4.8: Schematic view of converting relational data into targets

Target data are generated using a set of instance conversion rules. An algorithm has been developed for integrating the rules for each target database. The target data are generated in files as initial objects files and relationships files. Sets of customised SQL queries are embedded in these algorithms to extract the desired data from an RDB. Once a query is executed, the result is transformed from its RDB form into the target database format. Finally, a conversion program is generated to enact the schema file obtained from the schema translation phase and the files generated during the data conversion phase. The algorithms for data conversion are described in detail in Chapter 7.

Since relationships in object-based databases are represented by system-generated

identifiers, i.e., OIDs, defining such objects with their relationships in one-step conversion can lead to cross-referencing to objects that have not been created. Therefore, to achieve data integrity and establishing relationships consistently, the conversion process is accomplished in two separate passes. In the first pass, the tuples of each RDB relation comprising non-foreign key attributes are converted into the equivalent target format, in order to define the objects. Data of non-foreign key attributes are converted as literal attribute values of objects, or as an element or sets of elements. In the second pass, the initial objects defined in the first pass are linked using foreign key values extracted from each RDB relation's tuples based on the relationships defined in the target schema. The foreign key values, which realise relationships between tuples, are converted into value-based or user-defined identifers, which are called surrogate OIDs. Each surrogate OID is defined by concatenating its class name with the primary key values extracted from the corresponding RDB table. A surrogate OID is translated by the system into a physical OID during the creation of objects. Similarly, the object-based relationships are established using surrogates defined from the values of the CDM relationship attributes, i.e., $dirAs$ and $invAs$. However, relationships among XML elements are established by the `key`/`keyref` constraints specified in XML schema documents.

Example 4.5.1. Consider the CDM shown in Figure 4.5 and the RDB data given in Figure 4.4. One tuple from the `Salaried_emp` RDB table of an employee called 'Wallace', identified by the primary key value 54321, is converted with its related tuples in other tables into target equivalents. The output OODB object definition that represents the RDB 'Wallace' tuple is shown in Figure 4.9(a), whereas its relationships are defined in Figure 4.9(b).

(a)	`Salaried_emp54321 Salaried_emp (ename "Wallace", eno 54321, bdate "1931-06-20", address "91 St James Gate NE1 4BB", hasKids set(struct(kname "Scott", sex "M")), salary 43000);`
(b)	`salaried_emp54321->update()->projects.add(proj4);` `salaried_emp54321->update()->projects.add(proj5);`

Figure 4.9: Output OODB object definition and relationships

4.6 Summary

This chapter has provided an overview of the MIGROX solution to RDB migration. MIGROX is superior to the previous proposals as it produces three different output

databases. Besides, it exploits the range of powerful features that target data models provide such as ODMG 3.0, SQL4 and XML Schema.

The chapter started by introducing MIGROX and its three phases in Section 4.1, and then reviewed the assumptions on which the method is based in Section 4.2. Section 4.3 provided an introduction to the first phase of the solution, semantic enrichment, in which necessary data semantics about a given RDB are inferred and enhanced to produce an RSR. The RSR constructs are then classified to generate a CDM, which provides a description of the existing RDB's implicit and explicit semantics. An overview of the schema translation phase was given in Section 4.4. The CDM produced from the semantic enrichment phase is translated into any of the target schemas, applying an appropriate set of schema translation rules. Section 4.5 presented the data conversion phase, in which existing RDB data are converted into the format defined by the target schemas.

Chapter 5, the next chapter, describes in detail how to generate the CDM, as defined in Section 4.3.4, from the RSR and data stored in an existing RDB.

Chapter 5

Semantic Enrichment of Relational Database

In the previous chapter, an overview of the MIGROX solution was given. The first phase of MIGROX, the semantic enrichment, starts by obtaining a copy of an RDB metadata, to enhance its semantics and construct the RSR. The last step of this phase is to generate a CDM, which is appropriate for migration into the target databases. An algorithm has been developed to generate the CDM from RSR and existing RDB data. The goal of this chapter is to describe this algorithm, which allows the RSR constructs to be classified, enriched and translated into the CDM.

The rest of the chapter is structured as follows. Section 5.1 presents the algorithm for generating CDM. Sections 5.2 provides a summary of the chapter and points to what follows next.

5.1 Generation of CDM from RSR

Once the RSR has been derived from an existing RDB metadata, the interesting issue is mapping this representation into a CDM. This section presents the **GenerateCDM** algorithm as shown in Figure 5.1, which facilitates this mapping. Although the information provided by RSR is sufficient to begin generating the CDM, the focus here is on how to benefit from this information in order to identify and feed the CDM constructs, and to generate relationships and cardinalities among classes using data stored in the RDB.

Using key matching, RSR relations and their attributes are classified, relationships

1: **algorithm GenerateCDM** (rsr: RSR) **return** CDM
2: cdm: CDM := $\emptyset$ // a set to store CDM classes
3: **foreach** relation $R \in rsr$ **do**
4: C_{cdm}: $Class_{cdm}$ // define a CDM class as $\langle cn, cls, abs, A_{cdm}, REL, UK \rangle$
5: $C_{cdm}.cn := R.r_n$
6: $C_{cdm}.cls := classifyRelation(R)$
7: $C_{cdm}.A_{cdm} := classifyAttributes(R)$
8: $C_{cdm}.REL := identifyRelationships(R, C_{cdm})$
9: $C_{cdm}.abs := checkClassAbstraction(R)$
10: $C_{cdm}.UK := R.UK$
11: $cdm := cdm \cup \{C_{cdm}\}$ // add the class to CDM
12: **end for**
13: **return** cdm
14: **end algorithm**

Figure 5.1: The **GenerateCDM** Algorithm

between relations are identified and their cardinalities determined, followed by translation into equivalents in the CDM. The semantically enriched CDM forms the starting point for the remaining phases of the database migration process that leads to the generation of target schemas and then the conversion of relational data into target data. Given a set of RSR relation rsr as input, the algorithm goes through a main loop to classify each RSR relation $R \in rsr$ and its constructs, to generate their equivalents in the CDM (lines 3–12). The algorithm includes several functions for identifying and generating the CDM constructs. The *classifyRelation* function is used to classify R, with its attributes classified using the *classifyAttributes* function. Relationships are identified using the *identifyRelationships* function, while the *checkClassAbstraction* function is used to check whether the CDM class, mapped from R, is abstract or concrete. The set of unique keys UK remains unchanged. The following sections discuss these functions in detail.

Consider the RSR defined in Section 4.3.2, we assume the following functions and notations:

- $A(R)$ returns a set containing only a_n component (i.e., name) of attributes of relation R of type R_{rsr}.
- $P(R)$ returns a set containing only pa component (i.e., name) of the primary key attributes of R.
- $F(R)$ returns a set containing only fa component (i.e., attribute name) of all

foreign keys of R, while the function $fk(\beta)$ returns a set containing fa component of β, where $\beta \in R.FK$.

- $E(R)$ returns a set containing only ea component (i.e., attribute name) of all exported keys of R, while the function $ek(\gamma)$ returns a set containing ea component of γ, where $\gamma \in R.EK$.

- $numberOfDFKs(R)$ calculates and returns the number of disjoint foreign keys (DFKs) in the primary key of R. A composite primary key of a relation contains DFKs where parts of it are referencing two or more other relations.

- $numberOfDanglingKs(R)$ returns the number of dangling key attributes in R. A dangling key attribute is an attribute that is part of a composite primary key of a relation but not part of its foreign keys [Chiang et al., 1994].

- $getRSRrelation\ (r)$ returns the RSR relation that corresponds to the relation name r.

- $genCSS(X)$ generates a Comma Separated String from a set of attribute names X for projection in an SQL query.

- To get an element of a composite construct, we use '.' notation, e.g., $R.r_n$.

5.1.1 Identifying CDM Classes

An RSR relation is classified based on the comparison of its keys with that of other relations. Applying the *classifyRelation* function, shown in Figure 5.2, each RSR relation R (of type R_{rsr}) in the input set rsr (of type RSR) is classified based on arithmetic operations on its keys, and mapped into one of the nine CDM classes, defined in Section 4.3.4, according to the following rules:

- **Main relation**: The relation R is classified as RST if its primary key is not fully or partially composed of any foreign keys (i.e., $P(R) \cap F(R) = \emptyset$). However, R is classified as SST if it is a super-class (i.e., inherited by other classes). The regular strong class RST is distinguished from the super-class SST after all relationships of the class are identified as described in Section 5.1.3.

```
1: function classifyRelation (R: R_rsr) return cls
2:    classTag: cls // class classification
3:    A': set[attribute name] := A(R) − (P(R) ∪ F(R)) // a set of non-key attribute names
4:    if P(R) ∩ F(R) = ∅ then
5:       classTag := RST
6:    else if numberOfDFKs(R)> 1 then
7:       if numberOfDFKs(R) = 2 and A' = ∅ and E(R) = ∅ then
8:          classTag := RRC
9:       else
10:         classTag := SRC
11:      end if
12:   else if P(R) ⊆ F(R) then
13:      classTag := SUB
14:   else if F(R) − P(R) = ∅ and E(R) = ∅ then
15:      if numberOfDanglingKs(R) = 1 and A' = ∅ then
16:         classTag := MAC
17:      else
18:         classTag := CAC
19:      end if
20:   else
21:      classTag := RCC
22:   end if
23:   return classTag
24: end function
```

Figure 5.2: The *classifyRelation* Function

- **Relationship relation**: The relation R is a relationship relation if its primary key is composed fully or partially of two or more DFKs. A relationship relation has several forms: a binary M:N relationship relation with or without non-key attributes, or n-ary relationship relation where n>2. The function $numberOfDFKs$ is used to classify R into one of the two kinds of CDM classes as follows:

  1. R is classified as RRC if its primary key consists of two DFKs, R does not contain any non-key attributes ($A' := \emptyset$), and R is not referenced by other relations ($EK(R) := \emptyset$), or
  2. R is classified as SRC (i.e., its primary key consists of more than two DFKs, or $A' \neq \emptyset$ or $EK(R) \neq \emptyset$).

- **Sub-class relation**: The relation R is a sub-class relation and is classified as SUB, if its primary key is entirely composed of a primary key of another relation

(i.e., $P(R) \subseteq F(R)$) and all the pervious rules are not applicable. However, R is classified as SSC (i.e., a class in the middle of an inheritance hierarchy) if it is inherited by other relations. This distinction is preformed during the identification of relationships of the sub-class (see Section 5.1.3).

- **Weak relation**: The relation R is a weak relation if its primary key is partially composed of a primary key of another relation, i.e., parent relation, and none of the above rules is applicable. Thus, R is classified as follows:

  1. If R has no relationships with other relations except its parent relation (i.e., $F(R) - P(R) := \emptyset$ and $E(R) := \emptyset$), then:
     - R is a multi-valued attribute and is classified as MAC, if R has only one dangling key attribute and does not contain non-key attributes ($A' := \emptyset$). The function $numberOfDanglingKs(R)$ returns the number of dangling key attributes of R.
     - Otherwise, R is a composite attribute and is classified as CAC.
  2. R is a regular weak entity relation and is classified as RCC if it participates in one or more relationships with other relations in addition to its parent relation (i.e., $F(R) - P(R) \neq \emptyset$ or $E(R) \neq \emptyset$).

5.1.2 Identifying Attributes

Attributes of R are identified and mapped along with other properties into attributes of CDM class C_{cdm} using the *classifyAttributes* function shown in Figure 5.3. The function takes R as input and, from its attributes $R.A_{rsr}$, returns the set $classAtt$ (of type A_{cdm}) that contains all attributes of C_{cdm}. The properties of each attribute $att \in R.A_{rsr}$ include its name a_n, its data type t, length l, default d value, and whether or not it is nullable n ('y'|'n') (as defined in Section 4.3.4). In addition, att is classified, using $aTag$, into a non-key attribute 'NK', a primary key attribute 'PK', a foreign key attribute 'FK' or a primary-foreign key attribute 'PF'. These properties, combined as one 6-tuple for each single attribute, are added to the attributes of the CDM class.

```
1: function classifyAttributes (R: R_rsr) return A_cdm
2:    classAtt: A_cdm := ∅ // a set to store attributes of CDM class
3:    aTag: tag // attribute tag
4:    foreach attribute att ∈ R.A_rsr do
5:       if att.a_n ∈ P(R) ∩ F(R) then
6:          aTag := 'PF'
7:       else if att.a_n ∈ P(R) then
8:          aTag := 'PK'
9:       else if att.a_n ∈ F(R) then
10:         aTag := 'FK'
11:      else
12:         aTag := 'NK'
13:      end if
14:      classAtt := classAtt ∪ {⟨att.a_n, att.t, att.l, aTag, att.n, att.d⟩} // add one attribute
15:   end for
16:   return classAtt
17: end function
```

Figure 5.3: The *classifyAttributes* Function

5.1.3 Identifying Relationships and Cardinalities

This section presents the *identifyRelationship* function as shown in Figure 5.4. The function uses the key sets of each RSR relation $R \in rsr$ to generate the relationships of the corresponding CDM class C_{cdm}. Using information in the PK, FK and EK sets, the relationships among RSR relations are identified and classified, and their cardinalities determined, and they are then mapped into CDM relationships. For each R and the corresponding RDB data, every relationship in which R participates is identified and mapped into an equivalent CDM relationship and added to the set $aREL$ (of type REL) as an association, inheritance or aggregation, as defined in Section 4.3.4. The minimum and maximum cardinality *card* of each relationship is determined as *min..max* notation by querying the data in a complete database. The function *determinCard* determines *card* when R contains foreign keys, and the *determinInverseCard* function returns the inverse *card* when R is referenced by other relations. These two functions are also used to decide whether the relationship is optional or mandatory.

```
1: function identifyRelationships (R: R_rsr, C_cdm: Class_cdm) return REL
2:   aREL: REL := ∅ // a set to store relationships of a CDM class
3:   expR, impR: R
4:   card: relationship cardinality
5:   relShipTyp: relationship type
6:   r, ri, re: relation name
7:   FKey, EKey: set[attribute name] := ∅
8:   r := R.r_n
9:   foreach foreign key β ∈ R.FK do
10:    expR := getRSRrelation (β.er)
11:    re := expR.r_n
12:    FKey := fk(β)
13:    if FKey ⊈ P(R) or numberOfDFKs(R) ⩾ 2 then
14:      relShipTyp := 'associated with'
15:    else if P(R) ⊆ FKey then
16:      relShipTyp := 'inherits'
17:    else if FKey ≠ F(R) then
18:      relShipTyp := 'associated with'
19:    end if
20:    card := determinCard(r, FKey) // determines cardinality
21:    aREL := aREL ∪ {⟨relShipTyp, re, P(expR), card, FKey⟩} // add relationship
22:  end for
23:  foreach exported key γ ∈ R.EK do
24:    impR := getRSRrelation (γ.ir)
25:    ri := impR.r_n
26:    EKey := ek(γ)
27:    if EKey ⊈ P(impR) or numberOfDFKs(impR) ⩾ 2 then
28:      relShipTyp := 'associated with'
29:    else
30:      if EKey ≠ P(impR) then
31:        if EKey = F(impR) then
32:          relShipTyp := 'aggregates'
33:        else
34:          relShipTyp := 'associated with'
35:        end if
36:      else
37:        relShipTyp := 'inherited by'
38:        if C_cdm.cls = RST then
39:          C_cdm.cls := SST
40:        else
41:          C_cdm.cls := SSC
42:        end if
43:      end if
44:    end if
45:    card := determinInverseCard(r, ri, EKey) // determines inverse cardinality
46:    aREL := aREL ∪ {⟨relShipTyp, ri, EKey, card, P(R)⟩} // add relationship
47:  end for
48:  return aREL
49: end function
```

Figure 5.4: The *identifyRelationships* Function

Direct Relationships

The FK set of R shows relationships (i.e., '*associated with*', '*inherits*') when R has foreign keys (lines 9–22). Given that R participates in a relationship with an RSR relation $expR$ through β as a foreign key, where $\beta \in R.FK$. When R and $expR$ are mapped into corresponding classes, the relationship between them is mapped into a CDM relationship as follows:

- If the foreign key attributes of R are not part of its primary key attributes (i.e., $FKey \nsubseteq P(R)$) or if R contains two or more DFKs, then the relationship between the two classes is an association (i.e., the relationship type $relShipTyp$:= '*associated with*' - line 14).
- Otherwise, if the primary key attributes of R are a subset of its foreign key attributes (i.e., $P(R) \subseteq FKey$), then $relShipTyp$:= '*inherits*' - line 16.
- Otherwise, if R contains other foreign keys rather than those in $FKey$ (i.e., $FKey \neq F(R)$) then $relShipTyp$:= '*associated with*' - line 18.

After $relShipTyp$ is identified, the cardinality $card$ of the relationship is determined using the function *determinCard*, shown in Figure 5.5. The $genCSS$ function is used to generate a comma separated string from the set $FKey$ for projection in an SQL query from r, where r is the name of R (line 3 in Figure 5.5). The result of the query is assigned to T. If both T and r have the same number of tuples, then $card$:= 1..1 (i.e., the cardinality of the relationship is mandatory); otherwise $card$:= 0..1 (i.e., the cardinality is optional). Finally, a relationship in the form $\langle relShipTyp, re, P(expR), card, FKey\rangle$ is constructed and added to $aREL$, where re is the name of $expR$ and $P(expR)$ is a set containing the primary key attribute names of $expR$.

Inverse Relationships

The EK set helps to identify the inverse relationships (i.e., '*associated with*', '*aggregates*' and '*inherited by*'), when R is referenced relation (lines 23–47). Given that R participates in a relationship with an RSR relation $impR$ through γ as an exported key, where $\gamma \in R.EK$, and the function $P(impR)$ returns a set of primary key attributes of $impR$. When R and $impR$ are mapped into the corresponding classes, the relationship between them is mapped into a CDM relationship as follows:

```
1: function determinCard (r: relation name, FKey: set[attribute name]) return c
2:    card: c // cardinality
3:    T := execute('select '+genCSS(FKey)+' from '+r)
4:    if  size (r) = size (T) then
5:       card := 1..1 // check number of tuples
6:    else
7:       card := 0..1
8:    end if
9:    return card
10: end function
```

Figure 5.5: The *determinCard* Function

- If the exported key attributes of R are not part of the primary key attributes of $impR$ (i.e., $EKey \nsubseteq P(impR)$) or the primary key of R contains two or more DFKs, then $relShipTyp :=$ '*associated with*'.

- If the exported key of R and the primary key of $impR$ do not have the same attributes (i.e., $EKey \neq P(impR)$), then:

  If $impR$ has no other relationships (i.e., $EKey := F(impR)$), then $relShipTyp$:= '*aggregates*'; otherwise $relShipTyp :=$ '*associated with*'.

- Otherwise, $relShipTyp :=$ '*inherited by*'. By this relationship identified, if the CDM class C_{cdm} is classified as RST (a regular strong class), it will become a super-class SST, otherwise it will become an inherited sub-class SSC.

Given $relShipTyp$ identified, the cardinality $card$ of this inverse relationship is determined using the *determinInverseCard* function, shown in Figure 5.6. The $card$ is obtained by projecting ri on the attributes in $EKey$ and removing duplicates, where ri is the name of $impR$. The resulting set of tuples put in D. If the size of r is equal to the size of D (i.e., the relationship is mandatory), then if the size of D is less than the size of ri then $card :=$ 1..m, otherwise $card :=$ 1..1. However, if the size of D is less than the size of ri (i.e., the relationship is optional), then $card :=$ 0..m, otherwise $card :=$ 0..1. Finally, a relationship in the form $\langle relShipTyp, ri, EKey, card, P(R)\rangle$ is constructed and added to $aREL$.

```
1: function determinInverseCard (r, ri: relation name, EKey: set[attribute name]) return c
2:    card: c // cardinality
3:    D := execute('select distinct '+genCSS(EKey)+' from '+ri)
4:    if size(r) = size(D) then
5:       if size(D)< size(ri) then
6:          card := 1..m
7:       else
8:          card := 1..1
9:       end if
10:   else if size(D) < size(ri) then
11:      card := 0..m
12:   else
13:      card := 0..1
14:   end if
15:   return card
16: end function
```

Figure 5.6: The *determinInverseCard* Function

5.1.4 Identifying Class Abstraction

It is important for each super-class C_{cdm} (i.e., when $C_{cdm}.cls$:= (SST | SSC)) to be checked to determine whether it is a concrete or abstract class. The class is concrete (i.e., *abs* := `false`) when all (or some) of its corresponding RDB table rows are not members of its sub-tables. However, it is abstract (i.e., *abs* := `true`) when all of its objects are members of its sub-classes. Assume C'_{cdm} and C''_{cdm} are sub-classes of C_{cdm}, and $Ins(C_{cdm})$, $Ins(C'_{cdm})$ and $Ins(C''_{cdm})$ indicate the number of instances of C_{cdm}, C'_{cdm} and C''_{cdm}, respectively. Then, C_{cdm} is abstract if $Ins(C_{cdm}) := Ins(C'_{cdm}) + Ins(C''_{cdm})$, and is concrete class otherwise.

Example 5.1.1. Consider the RSR shown in Table 4.1 as input to the **Generate-CDM** algorithm, Table 5.1 shows (in part) the resulting CDM. Each relation in RSR is mapped into a class in CDM. For instance, the relation `Emp` is mapped into the CDM class `Emp`. The `Emp` class, which is an abstract SST class, has the attributes: *ename*, *eno*, *bdate*, *address*, *spreno* and *dno*. Other properties of the attributes (e.g., types, tags, length) are also shown. The class is '*associated with*' the classes: `Dept` (twice), `Works_on` and with itself (twice). Moreover, it '*aggregates*' the `Kids` class and is '*inherited by*' the `Salaried_emp` and `Hourly_emp` classes. Cardinality *c* and unique keys are also given for each class.

cn	*cls*	*abs*	A_{cdm}						*REL*					*UK*	
			a_n	*t*	*tag*	*l*	*n*	*d*	*relType*	*dirC*	*dirAs*	*c*	*invAs*	*ua*	*s*
Emp	SST	true	eno	int	PK	25	n		*asso*	Dept	dno	1..1	dno		
			ename	char		40	n		*asso*	Dept	mgr	0..1	eno		
			bdate	date			y		*asso*	Emp	eno	1..1	spreno		
			address	char		40	y		*asso*	Emp	spreno	0..*	eno		
			spreno	int	FK	25	y		*asso*	Works_on	eno	1..*	eno		
			dno	int	FK		n		*aggr*	Kids	eno	0..*	eno		
									inherBy	Salaried_emp	eno	1..1	eno		
									inherBy	Hourly_emp	eno	1..1	eno		
Salaried_emp	SUB	false	eno	int	PF	25	n		*inherts*	Emp	eno	1..1	eno		
			salary	int			y								
Dept	RST	false	dno	int	PK		n		*asso*	Emp	eno	1..1	mgr	mgr	1
			dname	char		40	n		*asso*	Emp	dno	1..*	dno		
			mgr	int	FK	25	n		*asso*	Proj	dnum	1..*	dno		
			startd	date			y		*aggr*	Dept_locations	dno	1..*	dno		
Works_on	RRC	false	eno	int	PF	25	n		*asso*	Emp	eno	1..1	eno		
			pno	int	PF		n		*asso*	Proj	pnum	1..1	pno		

asso: *associated with* *aggr*: *aggregates* *inherBy*: *inherited by*

Table 5.1: Results of CDM generation

5.2 Summary

This chapter has presented the **GenerateCDM** algorithm that generates the CDM from the RSR and RDB data. This is the last step of the semantic enrichment phase of MIGROX, where the first step was the construction of the RSR described in Section 4.3.3. This chapter has described the functions used in the algorithm, including classifying RSR constructs and generating their equivalents in the CDM. The algorithm classifies the RSR relations, their attributes and relationships using key matching and RDB data. The CDM provides a description of the existing RDB's implicit and explicit semantics. The CDM is then translated into any of the target schemas (in the second phase of MIGROX) and plays a key mediator rule for converting an existing RDB data into the target databases (in the third phase of MIGROX).

The next chapter describes how to translate the CDM into the target schemas (i.e., the second phase of MIGROX).

Chapter 6

Translation of CDM to Target Schemas

Chapter 5 explained the first phase of MIGROX, the semantic enrichment, in which an existing RDB is enhanced with explicit and implicit data semantics and mapped into the CDM. Once the CDM has been generated, then target schemas can be derived from it without any recourse to the source RDB. This chapter presents the second phase of MIGROX, the schema translation process, which was introduced in Section 4.4. This phase includes the translation of the CDM into OODB, ORDB and XML schemas. Three sets of rules for translating the CDM into each of the target schemas are proposed and illustrated using examples. Algorithms are developed for producing each target schema according to the relevant rules.

The chapter is organised as follows. Section 6.1 presents the common functions used by the algorithms. Translating CDM into an ODMG 3.0 ODL schema is provided in Section 6.2. Section 6.3 explains the mapping of CDM into an SQL4 ORDB schema, and mapping the CDM into an XML Schema is described in Section 6.4. Section 6.5 provides a summary of this chapter and points to what follows.

6.1 Common CDM Translation Functions

The following functions are used by the three algorithms described in this chapter:

- *mapAttrType*(*tdb*, *att*) translates the data type of a CDM attribute *att* into equivalent data type according the target database kind *tdb*, where *tdb* :=

('OODB' | 'ORDB' | 'XML'). Appendix A shows the mapping table used for attribute data type translation from CDM to OODB, ORDB and XML data types.

- *getCDMclass*($className$) returns the CDM class that corresponds to the class name $className$.

- *getRelationshipName* (x, X, y, Y) returns a concatenation of a class name x and a set of attribute names X with a class name y and a set of attribute names Y to derive a unique name for a relationship.

- *setMtoNrelationship*(*tdb*, *rel*). Given that an RDB cannot implement M:N relationships directly, this function resolves an M:N relationship into two 1:Ms. That is, if a CDM class C_{cdm} has a 1:M relationship *rel* (i.e., $rel \in C_{cdm}.REL$) with a class C'_{cdm}, where $C'_{cdm}.cls :=$ 'RRC', and C'_{cdm} has a 1:M relationship with a class C''_{cdm}, then depending on *tdb*, the function defines a new M:N relationship between the target classes corresponding to C_{cdm} and C''_{cdm}. This means that there will be no target class corresponding to C'_{cdm}.

- *mapNonFKtyp*(*tdb*, $C_{cdm}.A_{cdm}$) translates the data type of a non-foreign key attribute $att \in C_{cdm}.A_{cdm}$ defined in a CDM class C_{cdm} into a target equivalent type in *tdb*.

- *mapAttAndType*(*tdb*, $C_{cdm}.A_{cdm}$) takes the name and data type of every non-foreign key attribute $att \in C_{cdm}.A_{cdm}$ and translates them into the equivalent target types in *tdb* in the form of a structured type, e.g., `struct` when *tdb* := 'OODB' or `row` when *tdb* := 'ORDB'.

6.2 Translating CDM into OODB Schema

Given the CDM as defined in Section 4.3.4 and the ODMG 3.0 ODL as defined in Section 4.4.1, this section explains how the CDM is translated into an equivalent target OODB schema. A set of rules have been designed for translating CDM classes into corresponding ODL constructs. The **ProduceOODBschema** algorithm shown in Figure 6.1 implements these translation rules. Each rule is aimed at a specific construct, e.g., attribute translation, which may be supported by mapping functions, e.g., *mapAttrType*.

```
1: algorithm ProduceOODBschema (cdm: CDM) return OOschema
2:   targetSchema: OOSchema := ∅ // a set to represent the OODB target schema
3:   foreach class C_cdm ∈ cdm do
4:     if C_cdm.cls ≠ ('MAC' | 'CAC' | 'RRC') then
5:       C_oo: Class_oo // define an OODB class as ⟨c_n, spr, k, A_oo, REL_oo⟩
6:       mlt: string := 'sv'
7:       rel_nm, invRel_nm: string := '' // relationship names
8:       C_oo.c_n := C_cdm.cn
9:       C_oo.k := defineClassKey(C_cdm.A_cdm, C_cdm.cls)
10:      foreach attribute att ∈ C_cdm.A_cdm do
11:        if att.tag ≠ ('FK' | 'PF') then
12:          C_oo.A_oo := C_oo.A_oo ∪ {⟨att.a_n, mapAttrType('OODB', att), mlt⟩}
13:        end if
14:      end for
15:      foreach relationship rel ∈ C_cdm.REL do
16:        C'_cdm: Class_cdm := getCDMclass(rel.dirC)
17:        rel_nm := getRelationshipName (rel.dirC, rel.dirAs, C_cdm.cn, rel.invAs)
18:        if rel.c = (0..1 | 1..1) then
19:          mlt := 'sv'
20:        else
21:          mlt := 'cv'
22:        end if
23:        if rel.relType = 'associated with' then
24:          if C'_cdm.cls = 'RRC' then
25:            C_oo.REL_oo := C_oo.REL_oo ∪ {setMtoNrelationship('OODB', rel)}
26:          else
27:            invRel_nm := getRelationshipName (C_cdm.cn, rel.invAs, rel.dirC, rel.dirAs)
28:            C_oo.REL_oo := C_oo.REL_oo ∪ {⟨rel_nm, rel.dirC, mlt, invRel_nm⟩}
29:          end if
30:        else if rel.relType = 'aggregates' then
31:          if C'_cdm.cls = 'MAC' then
32:            C_oo.A_oo := C_oo.A_oo ∪ {⟨rel_nm, mapNonFKtyp('OODB', C'_cdm.A_cdm), mlt⟩}
33:          else
34:            C_oo.A_oo := C_oo.A_oo ∪ {⟨rel_nm, mapAttAndType ('OODB', C'_cdm.A_cdm), mlt⟩}
35:          end if
36:        else if rel.relType = 'inherits' then
37:          C_oo.spr := rel.dirC
38:        end if
39:      end for
40:      targetSchema := targetSchema ∪ {C_oo} // add the class to OODB schema
41:    end if
42:  end for
43:  return targetSchema
44: end algorithm
```

Figure 6.1: The **ProduceOODBschema** Algorithm

6.2.1 Translating Classes

For each CDM class $C_{cdm} \in cdm$, where $C_{cdm}.cls \neq$ ('MAC' | 'CAC' | 'RRC'), an OODB class C_{oo} (of type $Class_{oo}$) is created with its own properties, and added to the target schema $targetSchema$. The properties of C_{oo}, extracted from C_{cdm}, include its key k, super-class name spr, attributes A_{oo}, and relationships REL_{oo}, whereas the name of C_{oo} is assigned to be that of C_{cdm}. The C_{oo} is defined under its super-class via spr if $C_{cdm}.cls :=$ ('SUB' | 'SSC') as described in translating inheritance. The k is specified for strong classes, i.e., $C_{cdm}.cls :=$ ('RST' | 'SST') using the *defineClassKey* function (line 9), when one or more attributes of C_{cdm} are tagged with 'PK'. Sub-classes inherit their super-class keys. However, the OODB class must have an 'extent' to have a key. In ODMG 3.0, an extent defines the set of all instances of a given OODB class. The class name is appended by the letter '`s`' to represent the class extent, e.g., `Emps`.

Translating Atomic Attributes

Attributes are mapped according to two rules: a basic rule and a complex rule. The basic rule applies to the primitive attribute types. Each CDM attribute $att \in C_{cdm}.A_{cdm}$, where $att.tag \neq$ ('FK' | 'PF') is translated into an equivalent OO attribute and placed in the attribute set A_{oo} of C_{oo} with the same name as that of att using the *mapAttrType* function (lines 10-14). The complex rule has been designed for mapping aggregation relationships, resulting in simple/composite multi-valued attributes as described in translating aggregations.

Translating OODB Relationships

Each relationship rel defined in a CDM class C_{cdm} ($rel \in C_{cdm}.REL$) is translated into an equivalent relationship in the corresponding OODB class C_{oo} (lines 15-39). The type of the target relationship and its multiplicity are determined by the classification of the CDM class C'_{cdm} (i.e., $C'_{cdm}.cls$) related to C_{cdm} and the properties of rel (e.g., $rel.relType$). OO association, aggregation and inheritance relationships are derived from rel when $rel.relType :=$ '*associated with*', '*aggregates*' and '*inherits*', respectively. Moreover, the cardinality c of rel is mapped into a single-valued multiplicity '*sv*' when $c :=$ (0..1 | 1..1), or a collection-valued multiplicity '*cv*' otherwise.

- **Association**: Each relationship $rel \in C_{cdm}.REL$, where $rel.relType$:= '*associated with*', is translated into a corresponding bi-directional OO association relationship (lines 23-29). The relationship is represented by a pair of inverse references to ensure navigation in both directions and to preserve the referential integrity constraints. The direct relationship name rel_{nm} of the new relationship and also its inverse relationship name $invRel_{nm}$ are obtained using the *getRelationshipName* function, as described above. The rel is mapped into a single-valued '*sv*' relationship referencing the related OO class, corresponding to C'_{cdm} if $rel.c := (0..1 \mid 1..1)$, and into a collection-valued '*cv*' relationship if $rel.c := (0..* \mid 1..*)$. However, if $C'_{cdm}.cls :=$ 'RRC', then rel is mapped into an M:N relationship using the *setMtoNRelationship* function, in which a pair of 1:M relationships is defined in each of the OODB classes corresponding to the two CDM classes that participate in the relationship with C'_{cdm} (lines 24-25).

- **Aggregation**: Each relationship $rel \in C_{cdm}.REL$ where $rel.relType :=$ '*aggregates*' is translated, in C_{oo}, into an OO literal attribute (lines 30-35). The type of the attribute is mapped from the component C'_{cdm} that participates in the relationship with C_{cdm}, whereas its multiplicity mlt is derived from $rel.c$. The classification of C'_{cdm} determines the type of the attribute. When $C'_{cdm}.cls :=$ 'MAC', then rel is mapped into a multi-valued attribute, where its type is obtained by the *mapNonFKtyp* function. However, if $C'_{cdm}.cls :=$ 'CAC', then rel is mapped into a composite attribute, the type of which is returned using the *getAttAndType* function, as a (set of-) `struct` depending on $rel.c$

- **Inheritance**: Each relationship $rel \in C_{cdm}.REL$ where $rel.relType :=$ '*inherits*' is mapped into a simple inheritance, by which the sub-class C_{oo}, mapped from C_{cdm}, inherits all of the properties of its super-class mapped from C'_{cdm} (lines 36-37). The super-class name $C_{oo}.spr$ is assigned to $C'_{cdm}.cn$ to realise the inheritance between C_{oo} and its super-class. Additional attributes and relationships of C_{cdm} are mapped to C_{oo} in the usual way.

Example 6.2.1. Consider the CDM generated in Chapter 5 (shown in Table 5.1) as input to the **ProduceOODBschema** algorithm. Figure 6.2 shows the output ODMG 3.0 schema in ODL notation. The `Dept` and `Proj` classes are mapped as regular strong classes, whereas the `Emp` class is mapped as a super-class, from which the `Hourly_emp` and `Salaried_emp` classes inherit. Attributes and relationships are mapped from the CDM into the ODL schema. For instance, the OO class `Emp` has

the simple attributes: *ename*, *eno*, *bdate* and *address*. In addition, the class has many object-valued relationships with other classes (two relationships with `Dept`, one with `Proj` and two relationships with itself). The inverse directions of these relationships are shown in the figure. Moreover, the class aggregates `Kids` as a `struct` type. The multiplicity of all relationships are mapped for each class. The technique we follow in naming relationships is automatic, where the attributes that form the relationships are concatenated with the corresponding class names. For example, in CDM, the `Dept` and `Emp` classes participate in a relationship through the *mgr* attribute in the `Dept` class and *eno* attribute in the `Emp` class. The name of this relationship is mapped by concatenating these elements as a string separated by '_', i.e., `dept_mgr_emp_eno` (`manages`). Conversely, the inverse relationship name will be `emp_eno_dept_mgr` (`manager`). The user is allowed to change these names into more appropriate names.

```
class Emp (extent Emps key eno) {
   attribute string ename; attribute float eno;
   attribute date bdate; attribute string address;
   attribute set<struct Kids{string kname; string sex;}> hasKids;
   relationship Dept manages inverse Dept::manager;
   relationship set<Emp> supervises inverse Emp::supervisor;
   relationship Dept dept inverse Dept::employees;
   relationship Emp supervisor inverse Emp::supervises;
   relationship set<Proj> projects inverse Proj::employees;};

class Hourly_emp extends Emp (extent Hourly_emps){attribute long pay_scale;};
class Salaried_emp extends Emp (extent Salaried_emps){attribute long salary;};

class Dept (extent Depts key dno) {
   attribute string dname; attribute long dno; attribute date startd;
   attribute set<string> locations;
   relationship set<Emp> employees inverse Emp::dept;
   relationship set<Proj> controls inverse Proj::controledBy;
   relationship Emp manager inverse Emp::manages;};

class Proj (extent Projs key pnum) {
   attribute string pname; attribute long pnum; attribute string plocation;
   relationship set<Emp> employees inverse Emp::projects;
   relationship Dept controledBy inverse Dept::controls;};
```

Figure 6.2: Sample output OODB schema

6.3 Translating CDM into ORDB Schema

The ORDB schema that satisfies SQL4 as defined in Section 4.4.1 can be generated from the CDM. This section describes the **ProduceORDBschema** algorithm given in Figure 6.3, which returns the target schema. The algorithm produces the target ORDB SQL4 schema as a set of UDTs, a set of typed tables and a set of unique keys

stored in the sets aUT, aTT and aUK_{or}, respectively. The following sections explain the steps of the algorithm.

6.3.1 Creating User-Defined Types

To create typed tables for storing data it is necessary to define the underlying object types as UDTs. Each main CDM class $C_{cdm} \in cdm$ is translated into a UDT udt (of type $udType$). The name ut_n of udt takes the same name as that of C_{cdm}, i.e., $C_{cdm}.cn$, suffixed by a string '`_t`', e.g., `Emp_t`. Each udt is defined by deriving its attribute A_{ut} and identifying its super-type name s_{ut}. The udt is defined under its super-type via s_{ut} if $C_{cdm}.cls :=$ ('SUB' | 'SSC'). All UDTs, expect sub-class types, are defined with a self-referential attribute, using the '`ref using varchar(25)`' string, as part of their definitions. Tables defined based on those UDTs must then specify that the identifier $uoid$ of each object is user-generated. Once a UDT along with its properties is defined, then it is added to the set aUT, in which all UDTs are held.

Translating Atomic Attributes

Each non-foreign key attribute $att \in C_{cdm}.A_{cdm}$, i.e., $att.tag \neq$ ('FK' | 'PF') is translated into a primitive attribute in the target schema and added to the attribute set A_{ut} of udt (lines 14-18). Each target attribute $\in A_{ut}$ retains the same properties from att, i.e., $att.a_n$, $att.n$, $att.l$ and $att.d$, whereas its multiplicity mt is single-valued. The type of the target attribute is translated from att using the $mapNonFKtyp$ function, which takes $att.t$ and $att.l$ and returns the ORDB SQL4 equivalent type. Using this rule, all primitive attributes are mapped; however, the `row` and `ref`-based attributes are mapped as relationship attributes.

Translating ORDB Relationships

CDM relationships are translated into the target ORDB schema as association, aggregation and inheritance. Each relationship $rel \in C_{cdm}.REL$ is translated, based on $rel.relType$ and the classification of the related CDM class C'_{cdm}, into a relationship attribute and added into A_{ut} of the corresponding type, or mapped into an inheritance relationship (lines 19-43). The relationship attribute name rel_{nm} for association/aggregation relationships is generated using the $getRelationshipName$ function

```
1: algorithm ProduceORDBschema (cdm: CDM) return ORschema
2:   aUT: UT := ∅
3:   aTT: TT := ∅
4:   aUK_or: UK_or := ∅
5:   foreach class C_cdm ∈ cdm do
6:     if C_cdm.cls ≠ ('RRC' | 'MAC' | 'CAC') then
7:       udt: udtType // define a UDT as ⟨ut_n, s_ut, A_ut⟩
8:       T_or: tTable // define a typed table as ⟨tt_n, ut_n, s_tt, pk, uoid⟩
9:       mt, rel_nm, nl: string := ''
10:      udt.ut_n, T_or.ut_n := C_cdm.cn+'_t'
11:      T_or.tt_n := C_cdm.cn
12:      T_or.pk := definePKconstraint(C_cdm.cls, C_cdm.A_cdm)
13:      T_or.uoid := defineUserDefinedOID(C_cdm.cls) // specify uoid
14:      foreach attribute att ∈ C_cdm.A_cdm do
15:        if att.tag ≠ ('FK' | 'PF') then
16:          udt.A_ut := udt.A_ut ∪ {⟨a.a_n, mapAttrType('ORDB', att), mt, att.n, att.d⟩}
17:        end if
18:      end for
19:      foreach relationship rel ∈ C_cdm.REL do
20:        C'_cdm: Class_cdm := getCDMclass(rel.dirC) // get a related class
21:        rel_nm := getRelationshipName (rel.dirC, rel.dirAs, C_cdm.cn, rel.invAs)
22:        if rel.c = (0..1 | 1..1) then
23:          mt := 'sv'; nl := null/not null // depending on min cardinality of c
24:        else
25:          mt := 'cv'; nl := ''
26:        end if
27:        if rel.relType = 'associated with' then
28:          if C'_cdm.cls = 'RRC' then
29:            udt.A_ut := udt.A_ut ∪ {setMtoNrelationship('ORDB', rel)}
30:          else
31:            udt.A_ut := udt.A_ut ∪ {⟨rel_nm, 'ref('+rel.dirC+'_t)', mt, nl, ''⟩}
32:          end if
33:        else if rel.relType = 'aggregates' then
34:          if C'_cdm.cls = 'MAC' then
35:            udt.A_ut := udt.A_ut ∪ {⟨rel_nm, mapNonFKtyp('ORDB', C'_cdm.A_cdm), mt, nl, ''⟩}
36:          else
37:            udt.A_ut := udt.A_ut ∪ {⟨rel_nm, getAttAndType('ORDB', C'_cdm.A_cdm), mt, nl, ''⟩}
38:          end if
39:        else if rel.relType = 'inherits' then
40:          udt.s_ut := C'_cdm.cn+'_t'
41:          T_or.s_tt := C'_cdm.cn
42:        end if
43:      end for
44:      aUT := aUT ∪ {udt} // add the UDT to ORDB schema
45:      aTT := aTT ∪ {T_or} // add the table to ORDB schema
46:      aUK_or := aUK_or ∪ {defineUKconstraints(C_cdm.UK)}
47:    end if
48:  end for
49:  return ⟨aUT, aTT, aUK_or⟩
50: end algorithm
```

Figure 6.3: The **ProduceORDBschema** Algorithm

(line 21), whereas its multiplicity mt is single-valued 'sv' when $rel.c := (0..1 \mid 1..1)$, or collection '$cv$' when $rel.c := (0..m \mid 1..m)$ (lines 22-26).

- **Association**: Each relationship $rel \in C_{cdm}.REL$ where $rel.relType :=$ '*associated with*' is translated, in the udt corresponding to C_{cdm}, into an attribute where its type is a `ref` or a collection of `ref`s (depending on $rel.c$), referencing the related UDT mapped from the corresponding C'_{cdm} (lines 27-32). A `ref` attribute is constrained to be scoped (i.e., using a `scope` clause) to a specific table, so that the `ref` values stored in that attribute points to objects of the specified table. Each association is defined bi-directionally between its two types. However, unlike the OODB systems that support ODMG 3.0, neither SQL4 nor any ORDB products allow the user to rely on the system to automatically enforce inverse relationships. The inverse direction of the relationship is defined in the related UDT. Depending on the cardinalities c of rel and the classification of associated class $C'_{cdm}.cls$, rel is translated into 1:1, 1:M or M:N relationships. The rel is mapped, in udt, into a single-valued attribute of `ref` type, pointing to a pre-defined type corresponding C'_{cdm} (e.g., udt') when $rel.c := (0..1 \mid 1..1)$. Besides, rel is mapped into a collection-valued attribute, containing a collection of `ref`s, pointing to a related type udt' if $rel.c := (0..m \mid 1..m)$. However, rel is mapped into an M:N relationship if $C'_{cdm}.cls :=$ 'RRC' using the *setMtoNrelationship* function. As C'_{cdm} participates in only two M:1 association relationships with C_{cdm} and another CDM class C''_{cdm}, rel is mapped, in udt, into an attribute that contains a collection of `ref`s, pointing to a pre-defined udt'', corresponding to C''_{cdm}. Similarly, a collection of `ref`s, pointing to udt, is defined inside udt'' when mapping from the C''_{cdm} side.

- **Aggregation**: Each relationship $rel \in C_{cdm}.REL$ where $rel.relType :=$ '*aggregates*' is translated, in udt mapped from C_{cdm}, into a literal type attribute, representing C'_{cdm} (lines 33-38). Depending on c of rel and $C'_{cdm}.cls$, rel is translated into multi-valued attributes or `row` types. If $C'_{cdm}.cls :=$ 'MAC', rel is mapped into a multi-valued attribute, where the *mapNonFKtyp* function returns the data type of the attribute from $C'_{cdm}.A_{cdm}$. However, if $C'_{cdm}.cls :=$ 'CAC', rel is mapped into a `row` type attributes when $rel.c := (0..1 \mid 1..1)$, or as a collection of `row`s when $rel.c := (0..m \mid 1..m)$. The attributes of the `row` type are mapped from the non-foreign key attributes in $C'_{cdm}.A_{cdm}$ using the *getAttAndType* function.

- **Inheritance**: Each relationship $rel \in C_{cdm}.REL$ where $rel.relType :=$ '$inherits$' is mapped as a single inheritance, where *udt* translated from C_{cdm} inherits all of the properties of its super-type mapped from C'_{cdm}. The name of C'_{cdm} is appended by the string '_t' and assigned to super-type name attribute s_{ut} of *udt* to realise the inheritance, i.e., $udt.s_{ut} := C'_{cdm}.cn$+'_t' (line 40). Additional properties of *udt* are defined in the usual way. Creating a sub-type under its super-type is considered while creating the super-type by specifying the `not final` phrase at the end of the super-type definition, which is `final` by default. Specifying `not final` for a super-type in the `create type` statement means that other types can inherit from it.

6.3.2 Creating Typed Tables

A typed table T_{or} (of type $tTable$) is defined for each declared *udt* and labeled with the same name as the corresponding CDM class C_{cdm}, from which the *udt* has been translated, i.e., $T_{or}.tt_n := C_{cdm}.cn$. Creating T_{or} is based on UDT specifications, representing instances for each row in the table, i.e., $T_{or}.ut_n := C_{cdm}.cn$+'_t'. The primary key *pk* of T_{or} is defined using the *definePKconstraint* function, which produces *pk* from the attributes in $C_{cdm}.A_{cdm}$, where $tag :=$ 'PK' (line 12). Because T_{or} would contain objects that can be referenced by other objects, an identifier attribute $T_{or}.uoid$ is specified as user-generated OID to facilitate cyclic referencing among pre-created objects during data loading. When inserting a tuple in T_{or}, the *uoid*s of objects can be generated from primary key values of the corresponding RBD table. The function *defineUserDefinedOID* is used to define the *uoid* (line 13). However, sub-tables inherit primary keys and *uoid*s from their super-tables. Unique keys for each table are extracted from the CDM equivalents and placed in aUK_{or} using the function *defineUKconstraints* (line 46). Sub-tables are created `under` their super-tables via $T_{or}.s_{tt} := C'_{cdm}.cn$, where $C'_{cdm}.cn$ is the name of the corresponding super-class C'_{cdm} (line 41).

Example 6.3.1. Consider the CDM shown in Table 5.1 as input to the **Produce-ORDBschema** algorithm, Figure 6.4 shows the output SQL4 ORDB schema, which contains UDTs and typed tables. For example, the type `Emp_t` is created from the CDM class `Emp` and then used to create the `Emp` table. Non-foreign key attributes, e.g., *ename* and *eno* are mapped as simple attributes from the CDM, while other attributes define relationships such as `dept` (i.e., equivalent to the string `dept_dno_emp_dno` generated automatically) that references the type `Dept_t`. This attribute is translated

from the 1:1 association (i.e., ⟨'*associated with*', `Dept`, {*dno*}, 1..1, {*dno*}⟩) between the `Emp` and `Dept` CDM classes, which is defined in the `Emp` class and given in Table 5.1. In the inverse direction, a collection that contains `ref`s of `Emp_t` is defined in the `Dept_t` type to show that a set of employees work for a department. The `set`s are used to store a collection of values on the M side of relationships. The `Kids` class is mapped as a composite multi-valued attribute inside `Emp_t` using `set` and `row`, whereas the CDM `Dept_locations` class is mapped in `Dept_t` as a simple multi-valued attribute using `set`. Typed tables are created to store the actual data. Inheritance relationships among UDTs/tables are defined using the `under` phrase. `Hourly_emp_t` and `Salaried_emp_t` sub-types are mapped from the corresponding CDM classes and defined under the `Emp_t` super-type. The corresponding tables then become sub-tables of the `Emp` super-table, inheriting its properties.

```
create type Emp_t as (
   ename varchar(20), eno number, bdate date, address varchar(30),
   manages ref(Dept_t) scope Dept,
   supervises set(ref(Emp_t)),
   hasKids set(row(kname varchar(30), sex char(1))),
   projects set(ref(Proj_t)),
   dept ref(Dept_t) scope Dept,
   supervisor ref(Emp_t) scope Emp) not final
   ref using varchar(25);
create table Emp of Emp_t
   constraint Emp_pk primary key(eno), ref is uoid user generated;

create type Hourly_emp_t under Emp_t (pay_scale number) final;
create table Hourly_emp of Hourly_emp_t under Emp;
create type Salaried_emp_t under Emp_t (salary number) final;
create table Salaried_emp of Salaried_emp_t under Emp;

create type Dept_t as (
   dname varchar(20), dno number, startd date,
   locations set(varchar(25)),
   employees set(ref(Emp_t)),
   controls set(ref(Proj_t)),
   manager ref(Emp_t) scope Emp)
   ref using varchar(25);
create table Dept of Dept_t
   constraint Dept_pk primary key(dno), ref is uoid user generated;

create type Proj_t as (
   pname varchar(20), pnum number, plocation varchar(20),
   employees set(ref(Emp_t)),
   controledBy ref(Dept_t) scope Dept)
   ref using varchar(25);
create table Proj of Proj_t
   constraint Proj_pk primary key(pnum), ref is uoid user generated;
```

Figure 6.4: Sample output SQL4 ORDB schema

6.4 Translating CDM into XML Schema

An XML Schema file (`.xsd`) can be created based on the additional semantics captured in CDM. This section explains how to translate CDM constructs into an XML Schema. The translation process involves a set of mapping rules, which translate CDM into XML Schema annotations. An algorithm called **ProduceXMLschema**, given in Figure 6.5, has been developed for this purpose and represents an integration of these rules. The main steps of the algorithm are described in the following sections

6.4.1 Defining XML Namespaces

XML Schema documents have main components, e.g., complex type definitions, and secondary components, e.g., namespaces, annotations and language used. The secondary components must be defined in the first step in order to create an XML Schema document. A namespace is defined according to the standard for schema commands and assigned to a variable, e.g., `xs` as an XML Schema description using the attribute `xmlns` (XML namespace). All schema tags are prefixed by `xs:` to indicate the XML Schema namespace. However, the namespace can be defined as a default, where the use of such a prefix is not needed. Any necessary annotations used in the document have to be specified for the user and machine, e.g., English language (`xml:lang` = '`en`').

6.4.2 Declaring Schema Root and its Elements

An XML Schema is defined according to a tree data model, which has two main components. The first component is the declaration of the root element as a complex type that contains a sequence of elements and integrity constraints. The second component includes the definitions of the complex types of the elements declared under the root. Three common approaches are available to make the decision on how to define the schema's components, locally or globally. These approaches are Salami Slice, Russian Doll and Venetian Blind designs [Valentine et al., 2002]. Each approach can be adopted based on the application's requirements. In this research, the target XML Schema is produced according to the Venetian Blind design, which defines complex types globally and elements locally. This offers flexible component reusing and nest element declarations within type definitions.

```
1: algorithm ProduceXMLschema (cdm: CDM) return XMLSchema
2:   aRoot: Root // define the schema root
3:   aGT: GT := ∅ // a set to represent the complex types
4:   define the namespace, annotation and the aRoot' s name root_n
5:   nm, mx, rel_nm: string := ''
6:   foreach class C_cdm ∈ cdm do
7:     if C_cdm.cls ≠ ('MAC' | 'RRC') then
8:       ct: compType // define a complex type as ⟨ct_n, base, abst, LE⟩
9:       ct.ct_n := C_cdm.cn+'_t'
10:      ct.abst := C_cdm.abs
11:      foreach attribute att ∈ C_cdm.A_cdm do
12:        nm, mx := "1"
13:        if att.n = 'y' then
14:          mn := "0"
15:        end if
16:        if C_cdm.cls = ('CAC' | 'SUB' | 'SSC') then
17:          if att.tag ≠ 'PF' then
18:            ct.LE := ct.LE ∪ {⟨att.a_n, mapAttrType('XML', att), mn, mx⟩}
19:          end if
20:        else
21:          ct.LE := ct.LE ∪ {⟨att.a_n, mapAttrType('XML', att), mn, mx⟩}
22:        end if
23:      end for
24:      if C_cdm.cls ≠ 'CAC' and not C_cdm.abs then
25:        aRoot.PK_x := aRoot.PK_x∪ {definePK(C_cdm)}
26:        aRoot.FK_x := aRoot.FK_x∪ {defineFKs(C_cdm)}
27:        aRoot.UK_x := aRoot.UK_x∪ {defineUKs(C_cdm)}
28:        aRoot.LE := aRoot.LE ∪ {⟨C_cdm.cn, C_cdm.cn+'_t', "0", "unbounded"⟩}
29:        foreach relationship rel ∈ C_cdm.REL do
30:          C'_cdm: Class_cdm := getCDMclass(rel.dirC)
31:          if rel.relType = 'associated with' and C'_cdm.cls = 'RRC' then
32:            ct.LE := ct.LE ∪ {setMtoNrelationship('XML', rel)}
33:          else if rel.relType = 'aggregates' then
34:            rel_nm := getRelationshipName (rel.dirC, rel.dirAs, C_cdm.cn, rel.invAs)
35:            nm, mx := min and max cardinality rel.c
36:            if C'_cdm.cls = 'MAC' then
37:              ct.LE := ct.LE ∪ {⟨rel_nm, mapNonFKtyp ('XML' , C'_cdm.A_cdm), mn, mx⟩}
38:            else
39:              ct.LE := ct.LE ∪ {⟨rel_nm, rel_nm+'_t', mn, mx⟩}
40:            end if
41:          else if rel.relType = 'inherits' then
42:            ct.base := C'_cdm.cn+'_t'
43:          end if
44:        end for
45:      end if
46:      aGT := aGT ∪ {ct} //add the complex type to the global set aGT
47:    end if
48:  end for
49:  return ⟨aRoot, aGT⟩
50: end algorithm
```

Figure 6.5: The **ProduceXMLschema** Algorithm

After defining the namespace and annotations, the root element *aRoot* of the schema document is created and given a name $root_n$ with the same name as an existing RDB schema (or alternative provided by the user). The subsequent steps of the algorithm define the set of elements of the root $aRoot.LE$ and its identity-constraints PK_x, FK_x and UK_x, and then specify the set of global complex types aGT. The target XML Schema document is generated from *aRoot* and aGT. Each CDM main and concrete class $C_{cdm} \in cdm$ (i.e., $C_{cdm}.abs :=$ `false` and $C_{cdm}.cls \neq$ ('MAC' | 'CAC' | 'RRC')) is translated as an empty first-level element under the root – placed in $aRoot.LE$. Each element $\in aRoot.LE$ is named with the same name as the corresponding CDM class, i.e., $C_{cdm}.cn$ and has a type specified by adding the '_t' string to its name, i.e., $C_{cdm}.cn$+'_t'. The type name is used as a reference to a global complex type that is defined separately in the set aGT. The occurrence of each element is specified using the occurrence attributes $mn :=$ "0" and $mx :=$ "`unbounded`". The PK_x, FK_x and UK_x are defined for each element as described in Section 6.4.4.

6.4.3 Defining Complex Types

Each CDM class C_{cdm}, where $C_{cdm}.cls \neq$ ('MAC' | 'RRC') is translated into a global complex type *ct* (of type *CompType*) (lines 6-48). The name ct_n of *ct* is specified from the corresponding CDM class name $C_{cdm}.cn$, concatenated with the string '`_t`'. For example, the `Emp` class is mapped into the `Emp` element that has the `Emp_t` complex type. If C_{cdm} is abstract, i.e., $C_{cdm}.abs :=$ `true`, then *ct* is specified as abstract, i.e., $ct.abst := C_{cdm}.abs$. The set of elements $ct.LE$ is constructed from $C_{cdm}.A_{cdm}$ and $C_{cdm}.REL$. Each attribute $att \in C_{cdm}.A_{cdm}$ is translated into a local element and added to $ct.LE$ (lines 11-23). Each element is given a name as the same name of the corresponding *att*, and a type translated via the *mapAttrType* function. The occurrences *mn* and *mx* of the element, are set to default values where each is "1" since they are all of the primitive type. However, *mn* is set to "0" if *att* accepts nulls (i.e., $att.n :=$ 'y'). Foreign key attributes (i.e., tagged as 'FK' or 'PF') are defined in *ct* as normal simple attributes if they are specified in PK_x and FK_x sets; otherwise, they are dropped from the definition of *ct* (e.g., foreign key attributes in CAC classes). In other words, each attribute $att \in C_{cdm}.A_{cdm}$ are mapped into a local element, where $C_{cdm}.cls :=$ ('CAC' | 'SUB' | 'SSC') and $att.tag \neq$ 'PF'; whereas all attributes of C_{cdm}, where $C_{cdm}.cls :=$ ('RST' | 'RCC' | 'SRC') are mapped in *ct* into local elements. Other non-primitive elements are mapped from $C_{cdm}.REL$ as

described in Section 6.4.4.

6.4.4 Translating Relationships and Constraints

An XML Schema represents relationships among elements by specifying nested complex types, or constraints using the `key/keyref`. On the one hand, the definition of nested types follows the parent-child containment technique, which in most cases causes data redundancy, even though it may speed up the processing of queries by avoiding join operations. Moreover, nesting the elements under each other requires user help during the translation. On the other hand, defining relationships using `key/keyref` may result in a flat document, even though the documents generated using this technique contain less redundancy. Therefore, we follow each of these two techniques with the aim of producing less data redundancy in a nested document. Thus, relationships among main CDM classes are mapped into identity constraints using the `key/keyref`, whereas the MAC, CAC and RRC classes are translated as nested elements under their parent elements.

Identity Constraints: The sets PK_x, FK_x and FK_x are declared for candidate elements defined in *aRoot* from each corresponding CDM class C_{cdm} using $C_{cdm}.A_{cdm}$ and $C_{cdm}.REL$ (lines 25-27). The following functions are defined to return the three sets:

- *definePK*(C_{cdm}) returns the primary key pk for each element defined under *aRoot* in the form $\langle pk_n, selector, PKfield \rangle$, and adds it into the PK_x set. The pk element is translated from each attribute $att \in C_{cdm}.A_{cdm}$, where $C_{cdm}.cls$:= ('RST' | 'SRC' | 'RCC') and $att.tag$:= ('PK' | 'PF'). To guarantee the uniqueness, each key name pk_n is formed by concatenating $C_{cdm}.cn$ with each $att.a_n$ and the string 'PK'. A *selector* is assigned $C_{cdm}.cn$ as a constraint element scope, whereas $PKfield$ is specified from each $att \in C_{cdm}.A_{cdm}$ when $att.tag$:= ('PK' | 'PF') as related element(s) to the selected *selector*. The $PKfield$ can have more than one element in the case of a composite key. For example, the primary key *dno* of `Dept` class is translated into the XML primary key as $\langle$`"deptDnoPK"`, `Dept`, {`dno`}$\rangle$. However, if $C_{cdm}.cls$:= ('SUB' | 'SSC'), the corresponding set $PKfield$ contains the key attributes of the top super-type of the inheritance hierarchy.

- *defineFKs*(C_{cdm}) returns foreign keys for each element defined under *aRoot* in the form $\langle fk_n,\ ref,\ selector,\ FKfield \rangle$ and adds them to the FK_x set. XML foreign keys are mapped from each CDM relationship $rel \in C_{cdm}.REL$, where $rel.relType$:= '*associated with*', and the attribute names in $rel.invAs$, which are tagged as 'FK' or 'PF'. The foreign key name fk_n is formed by the concatenation of $C_{cdm}.cn$, with the name of each attribute in $rel.invAs$, and the string 'FK'. The $refer$ is formed from concatenating the $rel.dirC$, with the attribute names in $rel.dirAs$ and the string 'PK', whereas the *selector* is named as $C_{cdm}.cn$, and each element in $FKfield$ is assigned an attribute in $rel.invAs$.

- *defineUKs*(C_{cdm}) returns the unique keys for elements defined under *aRoot* and adds them to the set UK_x, based on their equivalents in CDM, i.e., $C_{cdm}.UK$.

Nested Elements: The following rules are used to translate CDM relationships into XML as sub-elements embedded within their parent elements (lines 29-44).

- Each association $rel \in C_{cdm}.REL$ between C_{cdm} and another CDM class C'_{cdm}, where $C'_{cdm}.cls$:= 'RRC', is translated using the *setMtoNrelationship* function into a multi-valued element in a complex type *ct* of the element translated from C_{cdm}. As C'_{cdm} participates in only two M:1 associations with C_{cdm} and another CDM class C''_{cdm}, *rel* is mapped in *ct* into a multi-valued element, referencing a complex type ct'', corresponding to C''_{cdm}. A foreign key is defined for the new sub-element through its parent element and added to the set FK_x, referencing the primary key of the element corresponding to C''_{cdm}. Similarly, a multi-valued element with its foreign key is defined in ct'', referencing *ct*.

- There are two different mapping rules that can be applied when $rel \in C_{cdm}.REL$ represents an aggregation relationship, i.e., where $rel.relType$:= '*aggregates*', between a parent class C_{cdm} and a component class C'_{cdm}. A sub-element corresponding to C'_{cdm} is defined and embedded in the parent complex type *ct*, mapped from C_{cdm}, representing this relationship. The name of the sub-element is generated by the *getRelationshipName* function, and its occurrences *mn* and *mx* are declared according to the corresponding cardinality *rel.c*. If $C'_{cdm}.cls$:= 'MAC', *rel* is mapped into a simple multi-valued sub-element, the type of which is extracted via the *mapNonFKtyp* function. However, if $C'_{cdm}.cls$:= 'CAC', *rel* is mapped into a multi-valued sub-element, the type of which is defined as a global complex type ct' and added into *aGT*, as described in Section 6.4.3.

Inheritance: Each relationship $rel \in C_{cdm}.REL$ defined in a class C_{cdm} that inherits another CDM class C'_{cdm} where $rel.relType :=$ '*inherits*', is mapped as an inheritance. A complex type ct corresponding to C_{cdm} is defined as an extension of its complex type ct' corresponding to C'_{cdm}. This realises an XML inheritance between ct and ct', where $ct.base := C'_{cdm}.cn$ (lines 41-42).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name= "XMLSchema">
<xs:complexType><xs:sequence>
      <xs:element name= "Dept" type= "Dept_t" maxOccurs = "unbounded"/>
      <xs:element name= "Hourly_emp" type= "Hourly_emp_t" maxOccurs = "unbounded"/>
      <xs:element name= "Salaried_emp" type= "Salaried_emp_t" maxOccurs = "unbounded"/>
      <xs:element name= "Proj" type= "Proj_t" maxOccurs = "unbounded"/>
   </xs:sequence></xs:complexType>
<xs:key name= "salaried_empEnoPK">
<xs:selector xpath= ".//Salaried_emp"/>
<xs:field xpath= "eno"/>
</xs:key>
...
<xs:keyref name= "projDnumFK" refer= "deptDnoPK">
<xs:selector xpath= ".//Proj"/>
<xs:field xpath= "dnum"/>
</xs:keyref>
...
</xs:element>
   <xs:complexType name= "Dept_t"><xs:sequence>
      <xs:element name= "dname" type= "xs:string"/>
      <xs:element name= "dno" type= "xs:int"/>
      <xs:element name= "mgr" type= "xs:int" minOccurs= "0"/>
      <xs:element name= "startd" type= "xs:date" minOccurs= "0"/>
      <xs:element name= "locations" type= "xs:string" maxOccurs= "unbounded"/>
   </xs:sequence></xs:complexType>
   <xs:complexType name= "Emp_t"> abstract= "true" <xs:sequence>
      <xs:element name= "ename" type= "xs:string"/>
      <xs:element name= "eno" type= "xs:int"/>
      <xs:element name= "bdate" type= "xs:date" minOccurs= "0"/>
      <xs:element name= "address" type= "xs:string"/>
      <xs:element name= "spreno" type= "xs:int" minOccurs= "0"/>
      <xs:element name= "dno" type= "xs:int"/>
      <xs:element name= "hasKids" type= "Kids_t" minOccurs= "0" maxOccurs= "unbounded"/>
      <xs:element name= "Projects" type= "Project_t" maxOccurs= "unbounded"/>
   </xs:sequence></xs:complexType>
   <xs:complexType name = "Salaried_emp_t"> <xs:complexContent>
      <xs:extension base= "Emp_t"> <xs:sequence>
         <xs:element name = "salary" type= "xs:int" minOccurs= "0"/>
      </xs:sequence> </xs:extension>
   </xs:complexContent></xs:complexType>
   <xs:complexType name = "Kids_t"> <xs:sequence>
      <xs:element name = "kname" type= "xs:string"/>
      <xs:element name = "sex" type= "xs:string" minOccurs= "0"/>
   </xs:sequence></xs:complexType>
...
</xs:schema>
```

Figure 6.6: Sample output XML Schema

Example 6.4.1. Figure 6.6 shows a portion of the XML Schema document generated from the CDM given in Table 5.1, according to the rules presented in Section 6.4.

The document illustrates the XML Schema constructs generated as a result of applying the **ProduceXMLschema** algorithm given in Figure 6.5. The CDM `Dept` class is translated into a local element `Dept` under the root `XMLSchema` and typed as `Dept_t`. The `Dept_t` is then defined globally in the schema document. Atomic attributes are mapped with `minOccurs/maxOccurs` occurrences from the corresponding ones in CDM, whereas association relationships and unique keys are mapped from the CDM into XML using the `key`, `keyref` and `unique` constraints. The `Kids` class is mapped into a complex type `Kids_t` that is referenced from the abstract complex type `Emp_t` using the attribute `hasKids` as a sub-element. The `Projects` is the name of a sub-element defined in `Emp_t` with appropriate type `Projects_t`. This sub-element is mapped from the `Works_in` CDM class, which represents an M:N relationship between `Emp` and `Proj`. The 1 side of these relationships is mapped into `minOccurs = "0"` when the relationship is optional, or is set to default (no mention of `minOccurs`) if it is mandatory. However, the M side is mapped as `maxOccurs = "unbounded"`. Inheritance among elements have been realised using `complexContent/extension` XML keywords. The `Hourly_emp_t` and `Salaried_emp_t` complex types are mapped from the equivalent CDM classes and defined under the `Emp_t` complex type. The corresponding elements then inherit properties from their super-type.

6.5 Summary

This chapter has described schema translation, the second phase of MIGROX. Three sets of translation rules have been designed for translating CDM into the equivalent target schemas according to the recognised standards. Algorithms have been developed for producing the target schemas according to the translation rules, which have been illustrated by suitable examples. The chapter started by presenting the common functions used by the algorithms in Section 6.1. Section 6.2 described the translation of the CDM into an ODMG 3.0 ODL schema. Section 6.3 then explained how to translate the CDM into an SQL4 ORDB schema, whereas mapping the CDM into an XML Schema was presented in Section 6.4. The CDM provides a basis for the mapping of an RDB into object-based and XML schemas, using a translation process employing mapping techniques specific to each of the target databases.

Converting RDB data into the target databases is described next in Chapter 7. The chapter explains in further detail how to extract, transform and load existing RDB data into the formats suitable for the target database systems.

Chapter 7

Conversion of Relational Data to Target Data

Schema translation, the second phase of MIGROX, has been presented in Chapter 6. Data conversion, introduced in Chapter 4, is the last phase of MIGROX. The aim of this phase is to convert existing RDB data into the target databases, in order to populate the schemas generated during the schema translation phase. For this purpose, three algorithms have been developed, which convert RDB data into the target data using CDM. This chapter explains how this process works.

This chapter is organised as follows. Section 7.1 introduces the common functions used in the algorithms. Section 7.2 presents converting relational data into OODB format. Section 7.3 describes converting relational data into ORDB format. Section 7.4 then explains the conversion of relational data into XML format. Section 7.5 concludes the chapter and points to what comes next.

7.1 Common Data Conversion Functions

The following are the common functions and definitions used by the algorithms described in this chapter.

- $getPK(C_{cdm})$, $getFK(C_{cdm})$ and $getNFK(C_{cdm})$ return respectively the primary key, foreign key and non-foreign key attribute names of a CDM class C_{cdm}.
- $getClassLeaves(C_{cdm})$ returns a list $classLeaves$, containing all sub-class names

of a super-class C_{cdm} ordered from bottom to top.

- *getCond*(C_{cdm}, *classLeaves*) returns an SQL `where` condition *cond*. The *cond* is added to the queries used in RDB data retrieval to exclude from a super-class RDB table T (corresponding to C_{cdm} where $C_{cdm}.cls$:= ('SST' | 'SSC')) all rows that are members in its sub-class tables, where names of those are stored in the *classLeaves* list. Through *cond*, the non-inherited RDB tuples in T are extracted and converted into the target database.

- *getClassHierarchy*(C_{cdm}) returns a list *classHierarchy*, containing all super-class names of a sub-class C_{cdm} ordered from top to bottom. During the generation of data of a target sub-class (e.g., C_{oo} corresponding to C_{cdm}), data from each RDB table corresponding to each of its top level super-classes stored in *classHierarchy* are also retrieved. Then, such data are converted into the target format in each of the target objects being defined or updated.

- *extractObjId*(x, *Data*) concatenates a class name x with a string of values stored in *Data*. This function is used to generate a user-defined object identifier for each object being defined.

- *getAttrMatchValues*(*Attrs*, *Values*) generates a string appended to a `where` clause, indicating that each attribute name in the set *Attrs* is equal to the corresponding value in the set *Values*. The string is used in an SQL selection/projection statement to specify that only certain rows of a table are retrieved.

- *extractValues*(t, *Attrs*) extracts values of attribute names in the set *Attrs* from a tuple t retrieved from an RDB.

7.2 Converting Relational Data into OODB

This section describes the **GenerateOOdata** algorithm, given in Figure 7.1, for extracting, converting and loading RDB data into files, which are then used to populate the OODB schema generated from the schema translation phase. The conversion process is performed in two passes. Section 7.2.1 presents the first pass, in which objects are defined to instantiate OODB classes with literal data. Section 7.2.2 describes the second pass which defines relationships among objects created in the first pass.

```
1: algorithm GenerateOOdata (cdm: CDM) return OODB data
2:   cond: string := ''
3:   foreach class C_cdm ∈ cdm do
4:     if not C_cdm.abs and C_cdm.cls ≠ ('MAC' | 'CAC' | 'RRC') then
5:       if C_cdm.cls = ('SST' | 'SSC') then
6:         cond := getCond(C_cdm, getClassLeaves(C_cdm))
7:       end if
8:       if C_cdm.cls ≠ ('SUB' | 'SSC') then
9:         instantiateOOclass(C_cdm, cond)
10:        estabOOclassAssocRel (C_cdm, cond)
11:      else
12:        classHierarchy := getClassHierarchy(C_cdm)
13:        instantiateOOsub-class(C_cdm, cond, classHierarchy)
14:        estabOOsub-classAssocRel(C_cdm, cond, classHierarchy)
15:      end if
16:    end if
17:  end for
18: end algorithm
```

Figure 7.1: The **GenerateOOdata** Algorithm

The algorithm takes each main and concrete CDM class $C_{cdm} \in cdm$ (i.e., $C_{cdm}.cls \neq$ ('MAC' | 'CAC' | 'RRC') and $C_{cdm}.abs :=$ `false`) to extract data from the RDB table T corresponding to C_{cdm}, and then converts such data to instantiate the corresponding OO class C_{oo}. However, data from T, where $C_{cdm}.cls \neq$ ('MAC' | 'CAC' | 'RRC') are converted as part of establishing aggregation and association relationships. The algorithm requires certain main functions to accomplish this process. Literal-valued attributes in C_{oo} are instantiated with their values from non-foreign key tuples in T. The non-foreign keys of T are obtained from the $getNFK(C_{cdm})$ function. However, foreign key attributes (i.e., obtained from the $getFK(C_{cdm})$ function), and CDM relationship attributes (i.e., $dirAs$ and $invAs$) are used to instantiate object-valued relationships.

The *instantiateOOclass* and *estabOOclassAssocRel* functions are used for instantiating OO non-sub-classes (i.e., each main class corresponding to C_{cdm} where $C_{cdm}.cls \neq$ ('SUB' | 'SSC')) with literal data and object-based relationships, respectively. However, although the *instantiateOOsub-class* and *estabOOsub-classAssocRel* functions work similarly; they are used to instantiate sub-classes differently, taking into consideration the properties (i.e., attributes and relationships) of data inherited from their top level super-class(es), where super-class names are stored in the $classHierarchy$

list. Several queries are embedded in these functions to obtain RDB tuples and store the results in sets, i.e., `ResultSets`. The `ResultSet` is a table of data representing a database result set, which is generated by executing queries on an RDB. Each tuple in a `ResultSet` is converted into an OO object definition or relationship definition in the format of object interchange format (OIF) files. The OIF files are then used to instantiate OODB classes using a specific program, i.e., `MakeFile`, to execute object files first and then relationship definition files.

The OIF is a specification language used to load and dump objects to/from an OODBMS using a file or a set of files [Cattell and Barry, 2000]. It supports all OODB concepts which comply with the ODMG 3.0 object model and schema definition. An OIF file contains object definitions that specify the type, attribute values, and relationships to other objects for each defined object. Each new object is defined by an object name, and a set of attribute values and relationship instances.

7.2.1 Instantiating OODB Classes

This section presents the *instantiateOOclass* function shown in Figure 7.2 for converting RDB data into OIF files. Data from each RDB table T corresponding to a main and concrete CDM class C_{cdm} are extracted and converted in order to instantiate the corresponding OO class C_{oo}. The function extracts data of selected attributes from each tuple in T to be converted into an OIF file named with $C_{cdm}.cn$.

1: **function** *instantiateOOclass*(C_{cdm}: $Class_{cdm}$, *cond*: SQL condition)
2: $queryString$, $uOID$, $objStruct$: string := ‘’
3: $queryString$:= $genCSS(getPK(C_{cdm}))$+‘, ’+$genCSS(getNFK(C_{cdm}))$
4: `ResultSet` := execute(‘`select` ’+$queryString$+‘ `from` ’+$C_{cdm}.cn$+$cond$)
5: **foreach** tuple $t \in$ `ResultSet` **do**
6: $setVal$: set[value] := $extractValues(t, getPK(C_{cdm}))$
7: $uOID$:= *extractObjId* ($C_{cdm}.cn$, $setVal$) // generates object name
8: $objStruct$:= $convObjAtomicAtt(extractValues(t, getNFK(C_{cdm})))$
9: **foreach** relationship $rel \in C_{cdm}.REL$ **and** $rel.relType$ = ‘*aggregates*’ **do**
10: $objStruct$+ := $convAggRel(C_{cdm}.cn, rel, setVal)$
11: **end for**
12: obj := *defineObject* ($C_{cdm}.cn, uOID, objStruct$) // generates OIF object definition
13: **end for**
14: **end function**

Figure 7.2: The *instantiateOOclass* Function

This is the first pass, where each target object *obj* of C_{oo} is generated by defining its user-defined object identifier $uOID$ and its structure $objStruct$, consisting of literal-based data. By definition, when an object is created, it is assigned a system generated OID, which is invisible to the user. The ODMG 3.0 standard allows objects to be named and supports the derivation of an OID from the name of the object, which is called an object tag name. Therefore, we use object names as $uOID$s to identify objects in OIF files. When $uOID$ and $objStruct$ are constructed, the *defineObject* function writes an OQL statement to an OIF file defining *obj*. However, object-valued relationships of *obj* are instantiated in the second pass as described in Section 7.2.2.

Extracting object identifiers: An SQL query that satisfies a particular condition *cond* is designed in order to retrieve primary key data (using $getPK(C_{cdm})$) and non-foreign keys data (using $getNFK(C_{cdm})$) from T and store the results in `SetResult` table. Then, from each tuple $t \in$ `SetResult`, the *extractObjId* function generates the $uOID$ for each *obj* by concatenating $C_{cdm}.cn$ with the data values of the primary key in t (extracted using $extractValues(t, getPK(C_{cdm}))$ function); thus the value of $uOID$ is guaranteed to be unique for each object, e.g., `'salaried_emp54321'`.

Instantiating literal-based atomic attributes: Data of non-foreign keys in t (extracted using $extractValues(t, getNFK(C_{cdm}))$ function) are converted to become the new OO atomic data of *obj*, and are assigned to $objStruct$ using the *convObjAtomicAtt* function.

Instantiating literal-based collections: For each CDM aggregation relationship $rel \in C_{cdm}.REL$ between C_{cdm} and a component class C'_{cdm}, where $rel.relType :=$ '*aggregates*' and $C'_{cdm}.cls :=$ ('MAC' | 'CAC'), the object *obj* is instantiated with literal-based collection data. The data are extracted from the RDB table T' corresponding to C'_{cdm} using the *convAggRel* function, shown in Figure 7.3. The function retrieves non-foreign key (using $getNFK(C'_{cdm})$) tuples from T' (i.e., $queryStr1$), where the set of relationship attribute(s) $dirAs$ in *rel* is equal to the primary key value retrieved from the parent table T (stored in $setVal$). Retrieved data are then stored in an `RSet` table, and restructured into the OIF format as a data collection $dataColl$. The attribute values in $dataColl$ are generated from non-foreign key tuples as normal scalar attributes. The OO multi-valued attribute data are generated when $C'_{cdm}.cls$

```
1: function convAggRel(className: string, rel: Rel, setVal: set[value]) return dataColl
2:   queryStr1, queryStr2, collV, dataColl, rel_nm: string := ''
3:   C'_cdm: Class_cdm := getCDMclass(rel.dirC)
4:   queryStr1 := genCSS(getNFK(C'_cdm))
5:   queryStr2 := getAttrMatchValues(rel.dirAs, setVal)
6:   rel_nm := getRelationshipName (rel.dirC, rel.dirAs, className, rel.invAs)
7:   RSet := execute('select '+queryStr1+' from '+C'_cdm.cn+' where '+queryStr2)
8:   foreach tuple tp ∈ RSet do
9:     if C'_cdm.cls = 'MAC' then
10:      dataColl+ := tp
11:    else
12:      foreach attribute att ∈ getNFK(C'_cdm) do
13:        collV+ := att+' '+extractValues(tp, att)
14:      end for
15:      dataColl+ := 'struct('+collV+')'
16:    end if
17:  end for
18:  if rel.c = (0..m | 1..m) then
19:    dataColl := 'set('+dataColl+')'
20:  end if
21:  dataColl := rel_nm+' '+dataColl
22: end function
```

Figure 7.3: The *convAggRel* Function

:= 'MAC', whereas `struct` type data are generated when $C'_{cdm}.cls$:= 'CAC'. The $dataColl$ is returned as a string which represents a collection, i.e., '`set`('+$dataColl$+')' when $rel.c$:= (0..m | 1..m), or as a single-valued attribute/`struct` otherwise. The $dataColl$ is then assigned to a corresponding relationship attribute rel_{nm} and appended to the $objStruct$ of obj being defined.

Realising inheritance among classes: Inheritance represents one of the main challenges during data conversion between databases that are fundamentally different. Unlike RDBs, object-based class structure in an inheritance hierarchy is inherited, whereas data should be placed in the leaves, i.e., sub-classes. Each OO sub-class C_{oo}, mapped from a CDM sub-class C_{cdm}, where $C_{cdm}.cls$:= 'SUB' (or $C_{cdm}.cls$:= 'SSC' if $C_{cdm}.abs$:= `false`) is instantiated with its own data from the corresponding RDB table T as described in Section 7.2.1. Besides, C_{oo} is instantiated by related data from the RDB tables corresponding to its top super-classes, which have their names stored in the $classHierarchy$ list. Through $classHierarchy$ all super-table(s) of C_{oo} are instantiated, realising the inheritance relationship. Consequently, the function(s) for

instantiating sub-classes are modified to take into account the *classHierarchy* list. Besides, the SQL queries in these functions are also redesigned accordingly to return data for each super-class table indexed by the primary key that exactly matches the primary key value of T corresponding to the sub-class C_{oo} being instantiated.

7.2.2 Establishing Object-valued OODB Relationships

After objects have been defined, it is necessary to establish the association relationships among them using their OIDs. This section explains the second step in the algorithm, which concerns updating pre-created OODB objects with relationships using the *estabOOclassAssocRel* function, shown in Figure 7.4.

The *estabOOclassAssocRel* function takes each CDM relationship rel defined in a CDM class C_{cdm}, where $rel \in C_{cdm}.REL$ and $rel.relType :=$ '*associated with*' in order to instantiate the corresponding direct relationship defined in C_{oo}, mapped from C_{cdm}. The direct relationship is identified when $rel.invAs \subseteq getPK(C_{cdm})$. The inverse relationships defined in the schema are enforced by the system in the ODMG 3.0 standard. The $uOID$ of each object being updated is extracted from the primary key data of each tuple t stored in `ResultSet` (extracted using $extractValues(t, getPK(C_{cdm}))$ function). Data in `ResultSet` are retrieved from the RDB table T corresponding to C_{cdm}. The direct relationships of C_{oo} is updated by the object identifiers t_uOIDs of target objects, related to the object being updated. The t_uOIDs are extracted from data of the primary key of the table T' corresponding to the CDM class C'_{cdm} related to C_{cdm}, and stored in a set called $t\_uOIDset$. The set $t\_uOIDset$ is constructed from one of the following:

1. From the tuples extracted by a projection on the primary key (extracted using $getPK(C'_{cdm})$) of the RDB table T', when $C'_{cdm}.cls \neq$ 'RRC', and the set of relationship attribute(s) $rel.dirAs$ in C_{cdm} equals the primary key values retrieved from T (extracted using $extractValues(t, getPK(C_{cdm}))$ function). Each t_uOID is extracted, using the *extractObjId* function, from each tuple tp stored in `RSet`, and then added to the $t\_uOIDset$.

2. From the tuples of a set of relationship attribute(s) $rel'.invAs$ retrieved from T', when $C'_{cdm}.cls :=$ 'RRC' applying the $establishMNrel(rel)$ function. The rel' is a CDM relationship that C'_{cdm} participates in with the other class C''_{cdm}. The

```
1: function estabOOclassAssocRel(C_cdm: Class_cdm, cond: SQL condition)
2:   queryStr1, queryStr2, queryStr3, rel_nm, uOID: string := ''
3:   t_uOIDset: set[uOID] := ∅ // a set of target uOIDs
4:   queryStr1 := genCSS(getPK(C_cdm))
5:   foreach relationship rel ∈ C_cdm.REL do
6:     if rel.relType := 'associated with' and rel.invAs ⊆ getPK(C_cdm) then
7:       C'_cdm: Class_cdm := getCDMclass(rel.dirC)
8:       T' := C'_cdm.cn
9:       queryStr2 := genCSS(getPK(C'_cdm))
10:      rel_nm := getRelationshipName (rel.dirC, rel.dirAs, C_cdm.cn, rel.invAs)
11:      ResultSet := execute('select '+queryStr1+' from '+C_cdm.cn+cond)
12:      foreach tuple t ∈ ResultSet do
13:        queryStr3 := getAttrMatchValues(rel.dirAs, t)
14:        setVal: set[value] := extractValues(t, getPK(C_cdm))
15:        uOID := extractObjId (C_cdm.cn, setVal)
16:        if C'_cdm.cls = ('SST' | 'SSC') then
17:          T' := getObjectClass(rel.invAs, t)
18:        end if
19:        if C'_cdm.cls = 'RRC' then
20:          t_uOIDset := establishMNrel(rel)
21:        else
22:          RSet := execute('select '+queryStr2+' from '+T'+' where '+queryStr3)
23:          foreach tuple tp ∈ RSet do
24:            t_uOIDset := t_uOIDset ∪ {⟨extractObjId(T', tp)⟩}
25:          end for
26:        end if
27:        updateRel (C_cdm.cn, rel_nm, uOID, t_uOIDset) // update a relationship
28:      end for
29:    end if
30:  end for
31: end function
```

Figure 7.4: The *estabOOclassAssocRel* Function

function allows the conversion of only one side of the M:N relationship, where the inverse direction is enforced automatically by the system.

When the $uOID$ and $t\_uOIDset$ for each object are extracted, the *updateRel* function generates an OQL statement in the OIF format, for updating the object relationship name rel_{nm} defined in C_{oo} by related t_uOIDs stored in the set $t\_uOIDset$. However, in the case that C'_{cdm} is a super-class, i.e., $C'_{cdm}.cls$:= ('SST' | 'SSC'), then the *getObjectClass* function is invoked to trace (using the *classLeaves* list) the target object that participates in the relationship being updated. This is to obtain the name of the required RDB table that contains the data of the relationship attributes

that match the data of attribute names $\in rel.invAs$ in t. The required t_uOID is then extracted by concatenating the RDB table's name with the data that represents the relationship.

Example 7.2.1. Consider the CDM generated in Chapter 5, shown in Table 5.1, and the RDB data given in Figure 4.4 as input to the **GenerateOOdata** algorithm. Figure 7.5 shows one OODB object converted from a tuple in the `Salaried_emp` RDB table of an employee called 'Wallace' identified by the primary key value 54321. The output object definition equivalent to 'Wallace' tuple is shown in Figure 7.5(a), whereas its relationships are defined in Figure 7.5(b). Data and relationships inherited from the super-class `Emp`'s object are shown.

(a)
```
Salaried_emp54321 Salaried_emp (ename "Wallace", eno 54321, bdate "1931-06-20", address "91
St James Gate NE1 4BB", hasKids set(struct(kname "Scott", sex "M")), salary 43000);
```

(b)
```
salaried_emp54321->update()->projects.add(proj4);
salaried_emp54321->update()->projects.add(proj5);
```

Figure 7.5: Output OODB data (OIF format)

7.3 Converting Relational Data into ORDB

ORDB data are managed and populated by instantiating UDTs and tables obtained from the schema translation phase. The process of converting RDB data into ORDBs results in text files, containing object/relationship data definitions. Having all files generated, they can be loaded into the ORDB system using a bulk-loading facility. Converting RDB data into ORDBs is similar to the process of converting RDB data into OODBs. The structure of the algorithm for generating ORDB data from an RDB is similar to the **GenerateOOdata** algorithm, presented in Section 7.2. Moreover, the conversion process here is also accomplished in two passes: typed tables are instantiated and then relationships among pre-defined objects are established. The main differences between the two algorithms lie in the resulting statements, which are used to define the target objects and relationships according to the standard supported by the target data model; for instance, `struct` type in ODMG 3.0 against `row` type in SQL4. In this section, we discuss only the differences between the two algorithms.

7.3.1 Instantiating Typed Tables

The population of a pre-created ORDB schema with initial object data is similar to the process described in Section 7.2.1. However, the generated object definitions for ORDB are written according to SQL4. In the first pass, each target object *obj* of a typed table T_{or} is generated by defining its user-defined object identifier $T_{or}.uoid$ and its structure $objStruct$, consisting of a literal-based data type. When *uoid* and $objStruct$ are constructed, the function writes a set of SQL statements to files, defining new instances of a type in each table using the UDT constructors. Object-valued relationships of *obj* are instantiated in the second pass.

Extracting object identifiers: An SQL query that satisfies a condition *cond* is designed in order to retrieve primary key and non-foreign key data from T corresponding to a CDM class C_{cdm} and store the results in `SetResult` table. Then, from each tuple $t \in$ `SetResult`, a user-defined identifier for each *obj* is generated by concatenating $C_{cdm}.cn$ with the values of the primary key of T in t. SQL4 allows self-referential attributes that can be user-defined as an identifier and specified as part of the type definition of the referenced table, by adding a `ref` clause to the `create table` statement (e.g., `ref is` *uoid* `user generated`). When the typed table is created, its *uoid* is specified as an additional column which stores the value of the identifier for each object in the table. The values of *uoid*s are then used in establishing relationships.

Instantiating literal-based atomic attributes: Data of non-foreign keys in t are converted to become the new ORDB atomic data of *obj* and are assigned to $objStruct$.

Instantiating literal-based collections: Each ORDB relationship attribute defined in typed table T_{or}, mapped from C_{cdm}, is instantiated by data based on the corresponding CDM aggregation relationship $rel \in C_{cdm}.REL$ between C_{cdm} and a component class C'_{cdm}, where $rel.relType :=$ '*aggregates*'. Data of non-foreign key attributes of RDB table T' corresponding to C'_{cdm}, where $C'_{cdm}.cls :=$ ('MAC' | 'CAC') are converted into ORDB data and cast to become a collection of literal/`row` data types. Such data retrieved from T', where $C'_{cdm}.cls :=$ 'MAC' are converted as multi-valued attribute data, e.g., `set`(value1, value2, ...). The `row` constructor is used to group the related instances converted from T', where $C'_{cdm}.cls :=$ 'CAC'. The SQL

`set` is used to cast the related collection of instances into a single data structure when $rel.c := (0..m \mid 1..m)$.

Realising inheritance among objects: Realising inheritance relationship among ORDB objects starts by instantiating each leaf object, in a sub-table, with its own data and data inherited from its root, i.e., top to bottom super-tables, names of which are stored in the *classHierarchy* list. Each ORDB sub-table T_{or} is instantiated by the data of its own attributes and relationships from the RDB table T corresponding to CDM sub-class C_{cdm}, where $C_{cdm}.cls :=$ 'SUB' (or $C_{cdm}.cls :=$ 'SSC' if $C_{cdm}.abs :=$ `false`) in the normal way for defining objects. In addition, data from the super-table(s) in the *classHierarchy* related to data from T are converted in order to instantiate ORDB super-table(s) via T_{or}, thus realising the inheritance relationship.

7.3.2 Establishing ref-based ORDB Relationships

After literal data have been generated, the second pass in the conversion process is to assign object identifers of pre-created objects to their relationship attributes. Unlike ODMG 3.0 standard, the inverse relationships are not enforced by the systems that support SQL4 standard. Therefore, the relationship attributes in ORDB schema need to be updated with the relationship data, through both sides of relationships. Object-valued relationships among objects are instantiated using their object identifers as `ref`s. The process results in a set of DML statements written in a file for updating each object in an typed table in order to instantiate its relationship attributes by target object identifer(s) of related objects.

Example 7.3.1. Consider the CDM shown in Table 5.1 and RDB data in Figure 4.4, Figure 7.6 shows the target ORDB object converted from the RDB tuple of the employee 'Wallace' identified by *eno* := 54321. Figure 7.6(a) shows a sample of ORDB SQL4 statement generated for instantiating the object, whereas Figure 7.6(b) shows the statements for updating the object with its relationships.

a)	`insert into Salaried_emp values (Salaried_emp_t('salaried_emp54321', 'Wallace', 54321, '1931-06-20', '91 St James Gate NE1 4BB', null, null, set(row('Scott', 'M')), null, null, null, 43000);`
(b)	`update Salaried_emp set manages = 'dept2', projects = set('proj4','proj5'), dept = 'dept2', supervisor = 'salaried_emp86655' where uoid = 'salaried_emp54321';`

Figure 7.6: Output ORDB SQL4 object definition

7.4 Converting Relational Data into XML

An XML instance document, i.e., `xml` file, can be generated from an RDB based on the CDM, in order to populate the XML Schema document, i.e., `xsd` file generated by the schema translation process. An XML schema document describes the structure of an XML document, and an XML instance document must be valid against the schema document. This section describes the **GenerateXMLdocument** algorithm, given in Figure 7.7, which inputs the CDM to extract and convert RDB data into an `xml` file that contains element instances and tags, which conform to the pre-defined XML schema document.

```
1: algorithm GenerateXMLdocument (cdm: CDM) return XML instance document
2:    create XML document and declare its namespace based on the XML Schema document
3:    define the root element name same as the one defined in the XML Schema document
4:    cond: string := ''
5:    foreach class C_cdm ∈ cdm do
6:       if not C_cdm.abs and C_cdm.cls ≠ ('MAC' | 'CAC' | 'RRC') then
7:          if C_cdm.cls = ('SST' | 'SSC') then
8:             cond := getCond(C_cdm, getClassLeaves(C_cdm))
9:          end if
10:         if C_cdm.cls ≠ ('SUB' | 'SSC') then
11:            generateXMLelement(C_cdm, cond)
12:         else
13:            generateXMLsub-classElement(C_cdm, cond, getClassHierarchy(C_cdm))
14:         end if
15:      end if
16:   end for
17: end algorithm
```

Figure 7.7: The **GenerateXMLdocument** Algorithm

The algorithm generates the target XML instance document applying functions, representing XML element instance conversion rules. A set of queries are embedded in these functions to obtain RDB tuples and store them in result set tables to be converted into XML element/sub-element instances and loaded into the `xml` file. After defining namespaces and the root of the document, the algorithm takes each concrete and main CDM class $C_{cdm} \in cdm$ (i.e., $C_{cdm}.abs :=$ `false` and $C_{cdm}.cls \neq$ ('MAC' | 'CAC' | 'RRC') and generates, from its corresponding RDB table, the instances of the XML element equivalent to C_{cdm}, using the *generateXMLelement* function. However, the target element is instantiated by data using the *generateXMLsub-classElement*

function when $C_{cdm}.cls$:= ('SUB' | 'SSC'), which takes into account the properties that the element inherits from its top level super-classes, the names of which are stored in $classHierarchy$. The following sections describe the steps of this algorithm in detail.

7.4.1 Defining Target Namespaces

XML declarations, including the definition of namespaces and the document root is generated according to the information defined in the schema document. To use a namespace in the instance document, it then has to be defined in the schema document using the `targetNamespace` attribute. However, a document instance could be defined without namespaces, in which case the `targetNamespace` is omitted from the schema document and it has to be indicated that the document instance does not define a namespace using `noNamespaceSchemaLocation` = `schemaDocuemt.xsd`, where `schemaDocuemt.xsd` is the name of the schema document. The XML Schema definition language (XSD) defines ceratin attributes which are used in XML instance documents[1]. For instance, the `noNamespaceSchemaLocation` attribute is defined in the instance document to associate a schema document that has no target namespace with an instance document.

7.4.2 Generation of Element Instances

The *generateXMLelement* function, given in Figure 7.8, is used to convert data from each RDB table T corresponding to a main and concrete CDM class C_{cdm} into an instance of an element defined under the root of the XML Schema. Instances of the element are generated from tuples in T based on the classification of C_{cdm} and related classes in the cdm. The converted data are loaded into an `xml` file after being reconstructed to fit the XML Schema data types and structures. Instances of the element are enclosed by two tags that are given the name of the corresponding C_{cdm}, i.e., $C_{cdm}.cn$. The conversion process starts by retrieving primary key data (i.e., using $getPK(C_{cdm})$) and all attribute names (stored the set $allAtt$) data from T, i.e., represented in the string $queryString$, with the results stored in `SetResult`. Attribute names in $allAtt$ are obtained from the corresponding attributes in $C_{cdm}.A_{cdm}$. From `SetResult`, data of attributes in $allAtt$ become atomic data of the new element

[1]These attributes can be found at: `http://www.w3.org/2001/XMLSchema-instance` namespace

```
1: function generateXMLelement(C_cdm: Class_cdm, cond: SQL condition)
2:   allAtt: set[attribute name] := getFK(C_cdm) ∪ getNFK(C_cdm) // all attributes in C_cdm
3:   queryString, atomicData, collectData: string := ''
4:   queryString := genCSS(getPK(C_cdm))+', '+genCSS(allAtt)
5:   ResultSet := execute('select '+queryString+' from '+C_cdm.cn+cond)
6:   foreach tuple t ∈ ResultSet do
7:     atomicData := convXmlAtomicAtt (extractValues(t, allAtt))
8:     setVal: set[value] := extractValues(t, getPK(C_cdm))
9:     foreach relationship rel ∈ C_cdm.REL do
10:      C'_cdm: Class_cdm := getCDMclass(rel.dirC)
11:      if rel.relType = 'aggregates' then
12:        collectData+ := establishXmlAggRel (C_cdm.cn, rel, setVal)
13:      else if rel.relType = 'associated with' and C'_cdm.cls = 'RRC' then
14:        collectData+ := establishXmlMNrel(C_cdm.cn, rel, setVal)
15:      end if
16:    end for
17:    defineElementInstance (C_cdm.cn, atomicData, collectData) // defines an element
18:  end for
19: end function
```

Figure 7.8: The *generateXMLelement* Function

and is appended to *atomicData*. Moreover, primary key data are used to generate multi-valued/composite data for any sub-elements that are defined in the element and appended to *collectData*. The data that formed *collectData* are converted from the RDB table T' corresponding to a CDM class C'_{cdm}, where C'_{cdm} participates in a relationship with C_{cdm} and $C'_{cdm}.cls :=$ ('MAC' | 'CAC' | 'RRC'). However, other relationships between the element being migrated and other elements are established by `key` and `keyref` constraints specified in the XML Schema document. From the data stored in *atomicData* and *collectData*, the function *defineElementInstance* writes an XML Schema code into the `xml` file defining instances of the element.

Generating data of atomic types: Applying the *convXmlAtomicAtt* function, each tuple t of the set attribute names *allAtt* stored in `SetResult` (extracted using $extractValues(t, allAtt)$ function) is converted to become an instance of a new atomic sub-element and appended to *atomicData*. Each such instance is enclosed by start and end tags that are named according to attribute name $att \in allAtt$.

Generating data of collection types: For each relationship rel that C_{cdm} participates in with a class C'_{cdm}, where $rel \in C_{cdm}.REL$ and $C'_{cdm}.cls :=$ ('MAC' |

'CAC' | 'RRC'), data from the RDB table T' corresponding to C'_{cdm} are converted as a collection $collV$ of sub-element instances in the parent element corresponding to C_{cdm}. This is where the relationship attribute(s) in $rel.dirAs$ is equal to the primary key value retrieved from the parent table T corresponding to C_{cdm} (extracted using $extractValues(t, getPK(C_{cdm}))$ function). Each instance of $collV$ is enclosed by start and end tags that are named according to $att \in getNFK(C'_{cdm})$, and then the values in $collV$ is enclosed by start and end tags that are named as a relationship attribute rel_{nm}, obtained from the *getRelationshipName* function. Retrieved data are converted into $collV$ for each sub-element in the parent element as multi-valued types and appended to $collectData$ using one of the following cases:

1. The function *establishXmlAggRel*, shown in Figure 7.9, retrieves non-foreign key (extracted using $getNFK(C'_{cdm})$) tuples from T', and converts them into a collection of sub-element instances, when $rel.relType :=$ '*aggregates*'.

2. The function *establishXmlMNrel* retrieves RDB data as a collection of sub-element instances to establish a binary M:N relationship between the parent element and another element, e.g., E''. When C_{cdm} participates in a relationship rel with a CDM class C'_{cdm}, where $rel.relType :=$ '*associated with*' and $C'_{cdm}.cls :=$ 'RRC', then the collection is generated from tuples in T' of the relationship attributes in $rel''.invAs$, where rel'' is the other relationship in which C'_{cdm} participates with the CDM class C''_{cdm}.

Realising inheritance among elements: Data from each RDB sub-table T corresponding to a CDM sub-class C_{cdm}, where $C_{cdm}.cls :=$ 'SUB' (or $C_{cdm}.cls :=$ 'SSC' when $C_{cdm}.abs :=$ `false`) are converted to instantiate an equivalent target XML sub-class element with its own data, in addition to data of its top level super-table(s) in the class hierarchy, where the name(s) of the super-class(es) are stored in the $classHierarchy$ list. Tuples from each super-class tables in $classHierarchy$, which are related to the tuples in T, are converted and defined through the sub-class element, realising the inheritance relationship. However, data of the primary key of T are not converted into the sub-class element since such data have already been inherited from its top super-class element.

```
1: function establishXmlAggRel(className: string, rel: Rel, setVal: set[value]) return
   collV
2:    queryStr1, queryStr2, collV, rel_nm: string := ''
3:    C'_cdm: Class_cdm := getCDMclass(rel.dirC)
4:    queryStr1 := genCSS(getNFK(C'_cdm))
5:    queryStr2 := getAttrMatchValues(rel.dirAs, setVal)
6:    rel_nm := getRelationshipName (rel.dirC, rel.dirAs, className, rel.invAs)
7:    RSet := execute(select '+queryStr1+' from '+C'_cdm.cn+' where '+queryStr2)
8:    foreach tuple tp ∈ RSet do
9:       foreach attribute att ∈ getNFK(C'_cdm) do
10:         collV+ := '<'+att+'>'+extractValues(tp, att)+'</'+att+'>'
11:      end for
12:      collV := '<'+rel_nm+'>'+collV+'</'+rel_nm+'>'
13:   end for
14:   return collV
15: end function
```

Figure 7.9: The *establishXmlAggRel* Function

Example 7.4.1. Consider the CDM shown in Table 5.1 and the RDB data given in Figure 4.4, Figure 7.10 shows a fragment of the XML document output of the employee 'Wallace' converted from its equivalent data in the RDB.

7.5 Summary

This chapter has described how to convert data from an RDB into the target databases. The chapter began by describing the common functions used by the algorithms in Section 7.1. Section 7.2 provided the instance conversion rules, describing the two passes for converting RDB data into the target OODB as object and relationship definitions. Section 7.3 presented the conversion of RDB data into ORDBs. RDB data conversion into object-based databases is accomplished in two passes in order to aid the consistent establishment of relationships. Apart from some differences related to the expressions of the standards of data definitions, the two algorithms for converting RDB data into object-based databases are very similar in their structure. The conversion of RDB data into XML documents is explained in Section 7.4.

Sets of queries are embedded in the conversion algorithms in order to obtain the desired RDB tuples, which are then converted based on the classification of CDM attributes and relationships. RDB data are extracted, converted and loaded into files,

```
<?xml version = "1.0" encoding = "UTF-8"?>
<XMLSchema xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation = "XMLSchema.xsd">
   ...
  <Salaried_emp>
     <ename>Wallace</ename>
     <eno>54321</eno>
     <bdate>1931-06-20</bdate>
     <address>91 St James Gate, NE1 4BB</address>
     <spreno>86655</spreno>
     <dno>2</dno>
     <hasKids>
          <kname>Scott</kname>
          <sex>M</sex>
     </hasKids>
     <projects>
          <pno>4</pno>
     </projects>
     <projects>
          <pno>5</pno>
     </projects>
     <salary>43000</salary>
  </Salaried_emp>
   ...
</XMLSchema>
```

Figure 7.10: Output XML instance document

which are then used to populate the target schema generated from the schema translation phase. The conversion processes were described with appropriate examples.

The next chapter, Chapter 8, discusses how to develop the prototype of MIGROX. The software used in its development and the architecture are depicted. In addition, the ways in which the prototype's components communicate with each other are explained.

Chapter 8

Implementation of the Prototype

The MIGROX solution for migrating RDBs into object-based and XML databases has been described in Chapters 4-7. The focus of this chapter is to describe how the solution is implemented as a prototype, involving the three phases of MIGROX, i.e., semantic enrichment, schema translation and data conversion. The input to the prototype is an RDB and the output is an object-based or XML database stored in files suitable for bulk loading into target platforms. The prototype contains encoded models as the implementation of the input and output databases, and algorithms and migration rules according to the concepts and assumptions described in this dissertation. The current prototype provides a basis for a proof of concept, and verifies that the concepts of MIGROX can be implemented in terms of programming languages.

This chapter is organised as follows. Section 8.1 discusses the development environment and the reasons behind choosing the software used in the implementation. The system architecture and the main modules of the prototype are illustrated in Section 8.2. Section 8.3 presents in detail the components of the prototype and how they work and communicate with each other. Section 8.4 then provides a summary of the chapter and points to what follows next in the dissertation.

8.1 Development Environment

The prototype of MIGROX is developed by implementing the algorithms described in Chapters 4-7. The algorithms were implemented using the Java 1.5.0 software development kit installed on a computer with CPU Pentium IV 3.2 GHz and RAM

1024 MB, operating under Windows XP Professional. The Java database connectivity (JDBC) API [Hamilton et al., 1997] has been utilised to establish a connection with an RDBMS, Oracle 11_g, which holds the input RDB(s) to be migrated.

8.1.1 Programming Language

Java, C#, C++ and Visual Basic are among the most popular programming languages these days. Each has its own distinctive characteristics. Although the implementation could be achieved using other programming languages, possibly with further efforts, Java [Charatan and Kans, 2005] was chosen for the development of the prototype. This is because Java is an OOPL, which is platform independent and supports software reuse. It allows efficient encoding of algorithms and is an compatible with many operating systems. In addition, several benchmarks have shown that Java has an acceptable level of performance compared to C/C++ [Lewis and Neumann, 2004]. Moreover, Java's JDBC API provides full access to the metadata of a given RDB and, using its metadata interfaces, can quickly retrieve a description of its tables and constraints from data dictionaries in the form of **ResultSet** objects [Hamilton et al., 1997].

8.1.2 Database Management System

Database systems store data and also information about these data, i.e., metadata. Most DBMSs have a set of system tables which define the basic structure of a database including table names, column names, constraints, etc. The Oracle DBMS has been chosen in the prototype as a back end. This is because Oracle provides the essential technologies for developing a range of applications, including Java applications. The benefits of Java applications include JDBC support. In addition, Oracle data dictionaries include numerous metadata tables and views, containing a large amount of information about table names, columns including names, data types, default values, primary keys, foreign keys, etc. The data dictionary can be accessed by JDBC and queried to retrieve this information.

8.1.3 Java Database Connectivity (JDBC)

JDBC is a set of Java APIs for executing SQL statements. These APIs consist of a set of classes and interfaces to enable programmers to access the data stored in databases. The JDBC API includes both the **java.sql** package (i.e., the JDBC core API) and the **javax.sql** package (i.e., the JDBC optional package API) [Hamilton et al., 1997]. The package **java.sql** provides the API that sends SQL statements to RBDs and retrieves the results of executing those SQL statements using the Java programming language. It contains many useful interfaces and methods, which have been used in the prototype, such as:

- Making a connection with a database via the **DriverManager**
  - **DriverManager** class: makes a connection with a driver.
  - **Driver interface**: represents a specific JDBC implementation for a particular database system.
- Sending SQL statements to a database
  - **Statement**: supports the execution of various kinds of SQL statements.
  - **Connection interface**: represents a connection to a database for retrieving and updating the results of a query, obtained with the **getMetaData** method.
  - **ResultSet interface**: this is a set of results returned by the database in response to SQL queries.
- Metadata
  - **DatabaseMetaData interface**: provides metadata about the database as a whole.
  - **ResultSetMetaData interface**: provides information about the columns of a **ResultSet** object.
- Exceptions

– **SQLException**: thrown by most methods when there is a problem in accessing data.

JDBC provides the interface **DatabaseMetaData**, which gives users and tools a standardised way to extract metadata. Some **DatabaseMetaData** methods return lists of information in the form of **ResultSet** [Hamilton et al., 1997]. Standard **ResultSet** methods, e.g., **getString** are used to extract data from objects in these **ResultSet**. We are concerned here with the following aspects of using the JDBC metadata facilities:

- Obtaining a list of tables available in the database.
- Obtaining information about the columns in those tables.
- Obtaining information about keys and unique constraints.

To accomplish this, several **DatabaseMetaData** methods have been used, which take arguments as string patterns. Those methods include the following:

- **getTables**: to get a description of tables available in a specified data dictionary.
- **getColumns**: to get a description of table attributes available in a data dictionary.
- **getPrimaryKeys**: to extract a description of the primary key attributes of a given table.
- **getExportedKeys**: to extract a description of the foreign key attributes that reference the given table's primary key (the foreign key exported by a table).
- **getImportedKeys**: to extract a description of the primary key attributes that are referenced by the given table's foreign key attributes (the primary keys imported by a table).

8.2 System Architecture

This section introduces the prototype that has been developed to demonstrate MIGROX. The prototype consists of three main modules: RDB Enricher, Schema Translator and Data Convertor. Figure 8.1 illustrates the prototype architecture and main

information flows among its modules and components. The user interacts with the system only to: 1) select the source RDB to be migrated; 2) select the kind of target database to be generated and choose whether to produce a schema only or a complete database; and 3) replace the relationship names generated automatically for target databases with more concise and manageable names.

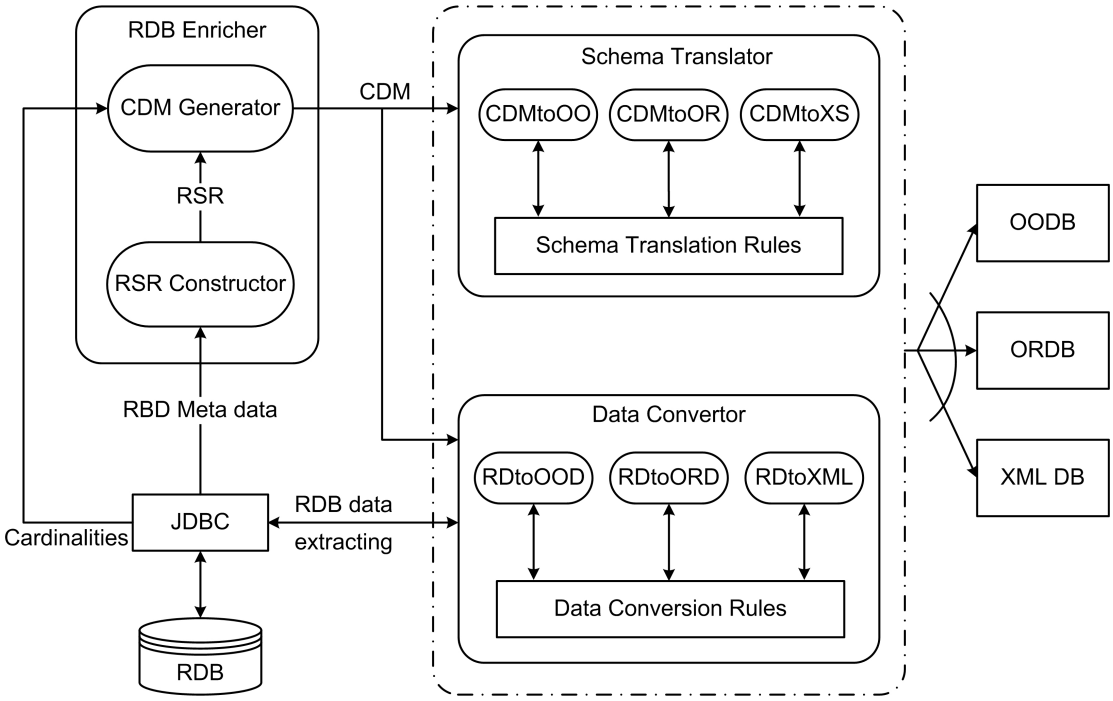

Figure 8.1: The overall architectural design

8.2.1 The RDB Enricher

The RDB Enricher has two main functions: constructing the RSR from RDB metadata and generating the CDM from the RSR and the data stored in an RDB. The module involves the implementation of the algorithms of the semantic enrichment phase, as described in Chapters 4 and 5. The RDB Enricher consists of two main components, an RSR Constructor and a CDM Generator.

For the first function, when the source RDB name has been determined by the user, the RDB Enricher connects to the RDBMS, using the JDBC/ODBC connection, and obtains a copy of the necessary RDB metadata information such as table names,

attributes and key constraints, and then returns an RSR structure. The RSR Constructor inputs each RDB table and produces the corresponding RSR relation, including its attributes and keys. For the second function, the RDB Enricher takes the RSR structure as input and accesses data stored in the RDB to produce the CDM. This process is conducted by the CDM Generator. During the identification of CDM relationships, the RDB Enricher automatically generates strings, which represent relationship names. After the CDM has been generated, the RDB Enricher interacts with the user to change these strings into the appropriate relationship names.

8.2.2 The Schema Translator

The CDM generated from the RDB Enricher is passed to the Schema Translator which translates it into the selected target schema after a restructuring according to the characteristics of the target database. The Schema Translator contains encoded schema translation rules integrated into the three schema translation algorithms described in Chapter 6. It implements the algorithms as three components, each of which concerns the production of one target schema. Each algorithm is implemented as one Java class consisting of a set of methods which represent the related rules. For each class in the CDM, the Translator translates its attributes and relationships into the corresponding constructs in the target database. The resulting schema is stored in a file.

8.2.3 The Data Convertor

The Data Convertor contains encoded instance conversion rules for the implementation of the data conversion algorithms, described in Chapter 7. The module converts data from a source database into any of the target databases based on the CDM. The Data Convertor extracts the source data from the RDBMS using dynamic SQL queries designed for this purpose. The data resulting from the execution of these queries are converted into the target data. The Data Convertor stores the converted data in directories that contain several script files for object definitions, relationship definitions and integrity constraints. The script files contain data definition statements suitable for loading into the target database systems.

8.3 Components of the Prototype

This section describes the components of the prototype modules introduced in Section 8.2. How these components work and communicate with each other is explained in more detail.

8.3.1 Components of the RDB Enricher

The RDB Enricher is responsible for acquiring a copy of the RDB metadata and enriching it with the required semantics in order to perform the rest of the phases of the database migration process. The RDB Enricher performs the enrichment in three main steps as described in Chapter 4. These steps, as shown in Figure 4.2, are: 1) the extraction of all possible information about tables, their attributes and keys from a given RDB; 2) the construction of an RSR; and 3) the generation of the CDM, which includes the classification of constructs and the identification of relationships and cardinalities. The two components of the RDB Enricher have been implemented as two main Java classes. Each of these classes includes a set of other class constructors and methods for the implementation of the data models and rules used during the enrichment process.

In order to construct the RSR, the required metadata stored in data dictionaries are obtained, using the packages and methods of JDBC metadata. Classifying RSR constructs and accessing data instances for the identification of relation and relationship types is the next step to be performed in generating the CDM. The obtained CDM is then an enhanced representation of the RDB ready to be translated into the target schemas, and to guide data conversion. After they are constructed the RSR and CDM can be displayed on the screen. The following subsections discuss the execution of these components.

The RSR Constructor

The function of the RSR Constructor is to obtain metadata information about an existing RDB, resulting in an RSR structure as implementation of the **ConstructRSR** algorithm described in Section 4.3.3. The algorithm is implemented as a main Java class `ConstructRSR`, which returns a vector called `RSR`, containing a set of objects of class called `RSRtable`. As defined in Section 4.3.2, the RSR is a set of relations,

where each relation R_{rsr} is represented as a 6-tuple $\langle r_n, A_{rsr}, PK, FK, EK, UK\rangle$, in which r_n denotes the name of R_{rsr}, A_{rsr} denotes a set of attributes of R_{rsr}, PK denotes the primary key of R_{rsr}, FK denotes a set of foreign keys of R_{rsr}, and so on. Therefore, the `RSR` represents the RSR and each object of `RSRtable` represents one R_{rsr} in the RSR, where `RSRtable` has a name r_n as well as five vectors representing the sets A_{rsr}, PK, FK, EK and UK as shown in Figure 8.2. The structures of these vectors are defined similarly.

```
public class RSRtable
 {
    private String tableName;
    private Vector A = new Vector();
    private Vector PK = new Vector();
    private Vector FKs = new Vector();
    private Vector EKs = new Vector();
    private Vector UKs = new Vector();

   // Constructors
   public RSRtable() {}
   public RSRtable(String tableNameIn){tableName = tableNameIn;}

   // accessor methods
   public String getTableName(){ return tableName;}
   public Vector getAttributes(){ return A;}
   public Vector getPK(){ return PK;}
   public Vector getFKs(){ return FKs;}
   public Vector getEKs(){ return EKs;}
   public Vector getUKs(){ return UKs;}

   // modifier methods
   public void setTableName(String tableNameIn){tableName = tableNameIn;}
   public void setPK (String tableNameIn, String columnNameIn, String keySeqIn, String pkNameIn)
     {PK.addElement(new PrimaryKey(tableNameIn, columnNameIn, keySeqIn, pkNameIn));}
   ...
}
```

Figure 8.2: The structure of the `RSRtable` class

Objects of `RSRtable` class are constructed using the methods invoked from the `ConstructRSR` class, which are then added to the `RSR` vector as the output of the RSR Constructor. Figure 8.3 shows a fragment of Java code for the `ConstructRSR` class with two of its methods, i.e., the `setPrimaryKey` and `setForiegnKeys`, which work as the constructors of the primary key and foreign keys of each relation in RSR, respectively.

The RSR Constructor interacts with the RDBMS and retrieves information about all of the tables in the RDB which have been selected for migration. From this information, the `RSR` vector is constructed. For each table name retrieved, its attributes and key constraints are obtained from the RDB. Attribute information includes column name, data type, length, precision, scale, null/not-null and default values. The

```
public ConstructRSR(){}
  Vector RSR = new Vector();
    ....
public ConstructRSR()
public Vector getAllTables(String url, String user, String password){
try {
    Connection con = DriverManager.getConnection(url, user, password);
    String [] types = {"TABLE"};
    DatabaseMetaData dbmd = con.getMetaData();
    String query1 = "select * from tab";
    Statement stmt1 = con.createStatement();
    ResultSet rss = stmt1.executeQuery(query1);
    while (rss.next()) {
      String tableName = rss.getString(1);
      RSRtable tableclass = new RSRtable(tableName);
        ....
      ResultSet rs1= dbmd.getPrimaryKeys(null, null, tableName);
      setPrimaryKey(rs1, tableclass);
      ResultSet rs2 = dbmd.getImportedKeys(null, null, tableName);
      setForiegnKey(rs2, tableclass);
        ....
      RSR.addElement(tableclass);
     }
    rss.close();
    con.close();
   }
  catch(SQLException ex){System.err.print("SQLException:"); System.err.println(ex.getMessage());}
  return RSR;
  }
  ....
public void setPrimaryKey(ResultSet rsIn, RSRtable tableclassIn) throws SQLException {
  while(rsIn.next()){
        String name = rsIn.getString("TABLE_NAME");
        String columnName = rsIn.getString("COLUMN_NAME");
        String keySeq = rsIn.getString("KEY_SEQ");
        String pkName = rsIn.getString("PK_NAME");
        tableclassIn.setPK(name, columnName, keySeq, pkName);
      }
}
  ....
public void setForiegnKeys(ResultSet rsIn, RSRtable tableclassIn) throws SQLException {
  while(rsIn.next()){
        String pkTable = rsIn.getString("PKTABLE_NAME");
        String pkColName = rsIn.getString("PKCOLUMN_NAME");
        String fkTable = rsIn.getString("FKTABLE_NAME");
        String fkColName = rsIn.getString("FKCOLUMN_NAME");
        short keySeq = rsIn.getShort("KEY_SEQ");
        String fkName = rsIn.getString("FK_NAME");
        String pkName = rsIn.getString("PK_NAME");
        tableclassIn.setFKs(fkTable, fkColName, keySeq, pkTable, pkColName, fkName, pkName);
      }
}
  ....
```

Figure 8.3: The structure of the `ConstructRSR` class

description of each column of a primary key, foreign key, exported key and unique key includes the key table name, key column, sequence number in case the key is a composite of more than one attribute, and the key constraint name. Inverse columns description of key columns that are linked to the given table's key columns are also retrieved as part of the description of the key being retrieved. Such information is obtained using the **DatabaseMetaData** methods of JDBC described earlier, such as **getTables()**, **getColumns()**, **getPrimaryKeys()**, **getExportedKeys()** and **getImportedKeys()**. When all of this information has been retrieved for one RDB table, an RSR relation is constructed and added into the RSR, i.e., `RSR` vector. Finally the RSR Constructor returns the RSR, which is then passed to the CDM Generator for generating the CDM.

The CDM Generator

When the RSR has been constructed, its relations and their attributes are classified, mainly based on the comparison of keys as described in Section 4.3.4, and then mapped into equivalents in CDM. In addition, relationships and cardinalities are identified and classified. This function is the responsibility of the CDM Generator, which implements the **GenerateCDM** algorithm described in Section 5.1. The CDM Generator takes the RSR as input and generates the CDM as output.

The **GenerateCDM** algorithm is implemented as a main Java class `GenerateCDM`, which returns a vector called `CDM`, containing a set of objects of a class `CDMclass`. The `CDM` vector represents the CDM and each object of `CDMclass` represents one class in the CDM. The `GenerateCDM` class contains a set of methods, each of which has a specific task, such as the classification of classes, checking whether or not a class is abstract, and mapping the type of attributes.

The CDM Generator classifies each relation in the RSR by matching its attributes, primary key, foreign keys and exported keys, and then maps the relation into one of the nine types of classes described in Section 4.3.4. Attributes are identified and classified into non-key attributes, primary key attributes or foreign key attributes using *tag*s. Relationships are identified and classified by matching the primary key, foreign keys and exported keys of each relation, while the unique keys are retained unchanged. Moreover, the CDM Generator accesses the RDB data to determine relationship cardinalities and to check whether super-classes are abstract or concrete

classes. Two methods are used to determine the cardinalities of relationships. The method `deterCard` determines cardinality when the RSR relation R_{rsr} contains foreign keys, and the method `deterInverCard` returns the inverse cardinality when R_{rsr} is referenced by other relations. The `checkAbstraction` method is invoked by the Generator when the abstraction of a CDM class needs to be checked. The method takes a CDM super-class as argument, accesses the RDBMS (for querying the corresponding RDB table and its sub-tables) and returns a variable `abs` of boolean type as a result. A super-class is not abstract (i.e., `abs` := `false`) when all (or some) of its corresponding RDB table rows are not members of other sub-table rows.

Example 8.3.1. Consider the CDM shown in Table 5.1 and the RDB data given in Figure 4.4. The following SQL query is used to check whether or not the CDM super-class corresponding to the `Emp` relation is abstract, where `Hourly_emp` and `Salaried_emp` are both its sub-class relations:

```
select count(*) from Emp e, Hourly_emp h, Salaried_emp s where e.eno =
h.eno(+) and e.eno = s.eno(+) and h.eno is null and s.eno is null;
```

If the query result is zero, then the `Emp` class is abstract; otherwise the class is a non-abstract (i.e., concrete) super-class.

From these identifications and classifications, the CDM Generator defines one CDM class and adds it into the CDM. The output of the Generator, the `CDM` vector, is stored as an in-memory representation to be used by the Schema Translator and Data Convertor, without having to repeatedly refer to the existing RDB.

During identification of the CDM association/aggregation relationships, the RDB Enricher automatically generates strings, which represent relationship names. Each string is generated as a concatenation of the names of classes which participate in the relationship and their attributes that form the relationships. After the CDM is generated, the RDB Enricher interacts with the user to change these strings and replace them with appropriately meaningful relationship names, e.g., according to the conceptual schema of the input RDB if available. Each string is stored as a pair with the new relationship name in a list called `relationshipNamesList`. The first element of the pair is the string generated by the Enricher and the second element of the pair is the name suggested by the user. This list is used later by the Schema Translator and Data Convertor to define relationships in the target schemas and data.

8.3.2 Components of the Schema Translator

The Schema Translator is responsible for translating the CDM generated by the RDB Enricher into the corresponding target schemas. This is performed by implementing the algorithms and the translation rules designed to produce the target schemas from the CDM, described in Chapter 6. Each set of rules concerning one target schema is implemented as one component as follows.

- **OODB Schema Mapper (CDMtoOO)**: The function of the CDMtoOO component is to translate the CDM into the corresponding OODB schema, implementing the **ProduceOODBschema** algorithm as described in Section 6.2. The OODB class definitions, including attributes and relationships are generated according to lambda-DB 1.8 [Fegaras, 2008], which supports ODMG 3.0 ODL specifications (but is not yet fully ODMG-compliant).

- **ORDB Schema Mapper (CDMtoOR)**: The function of the CDMtoOR component is to translate the CDM into the corresponding Oracle 11_g ORDB schema, implementing the **ProduceORDBschema** algorithm as described in Section 6.3. CDM classes are translated into object types, based on which object tables are defined when classes are non-abstract. As `row` and `set` constructs of SQL4 are not yet supported in Oracle 11_g, the M side of relationships and classes that represent multi-valued attributes are translated into nested tables.

- **XML Schema Mapper (CDMtoXS)**: The function of the CDMtoXS component is to translate the CDM into the corresponding XML Schema, implementing the **ProduceXMLschema** algorithm as described in Section 6.4. The main CDM classes are translated into complex types, based on which XML elements are defined when classes are non-abstract. Attributes that are classified as primary keys, foreign keys or unique keys are translated into XML using the key mechanism provided by the XML Schema language, i.e., `key`, `refkey` and `unique`, respectively. Null/not-null and relationship cardinalities are translated using `"minOccurs"` and `"maxOccurs"` keywords.

Each component of the Schema Translator is implemented as a main Java class, where its translation rules are encoded as methods. Each set of methods is embedded in the related class. Each method is responsible for performing a specific mapping task, such as mapping attributes and their types, and the translations of relationships.

The Schema Translator starts when the target schema/database to be produced is determined by the user. Then, the Translator calls up the constructor method of the appropriate component, which in turn applies the suitable set of methods (rules) for mapping the CDM constructs into their equivalent in the target schema. Using the classification of CDM constructs, the Translator identifies their equivalents in the target schema definition language. However, the Schema Translator also contains common methods, which can be used by any of its three components. For instance, the method `mapAttributeType`(*tdb*, *att*) (see Figure 8.4) takes the target database type *tdb*, e.g., `"OODB"` and a CDM attribute of type *att*, e.g., `tempClassAttribute` as parameters and returns the corresponding data type. Appendix A provides the RDB attribute data types and their equivalent data types in the target databases.

The way that the ORDB Schema Mapper and the XML Schema Mapper are implemented is similar to the implementation of the OODB Schema Mapper, apart from some differences, e.g., generating the constructs according to the target database schema. Therefore, we describe here only the OODB Schema Mapper.

The OODB Schema Mapper (CDMtoOO)

The CDMtoOO takes the CDM generated by the RDB Enricher and translates it into equivalent ODMG 3.0 ODL schema, applying appropriate set of rules. The schema translation rules of the CDMtoOO are encoded as four methods, which are responsible for mapping: atomic-valued attributes, multi-valued attributes, `struct` types and relationships. Each of these mappings is based on the classification of CDM constructs. After the mapping of the CDM into the target schema, the CDMtoOO then writes the generated schema into a file. Finally, the CDMtoOO generates a conversion program, called `MakeFile`. The `MakeFile` program is to enact the schema file first, and then the data files if generated by the RDtoOOD.

An example of mapping CDM attributes into equivalent atomic attributes in a target ODL schema is shown in Figure 8.4. Attributes are translated into equivalents with the same names as those of the CDM, and their types are converted according to target data types. Keys are specified using tags, e.g., `attributeTag` = `"PK"`. For this rule, the method `setClassAttributes` performs the mapping. The arguments of the method are the OODB class `ooClassIn` being translated and the vector `cdmAttributesIn` that contains a set of attributes of the CDM class, corresponding

to `ooClassIn`. The method goes through a loop for each object of the CDM attribute `tempClassAttribute` from the `cdmAttributesIn` vector, and maps them one by one. The `tempClassAttribute` is translated into an OODB attribute if it is a non-foreign key attribute, i.e., `attributeTag` $\neq$ (`"FK"` | `"PKFK"`) (foreign keys are mapped to relationships). The type of the attribute is mapped using the `mapAttributeType` method. Finally, the new attribute is added to the attributes of `ooClassIn`.

```
private void setClassAttributes (OOClass ooClassIn, Vector cdmAttributesIn)
 {
   for(int j = 0; j<cdmAttributesIn.size(); j++)
     {
       CDMClassAttributes tempClassAttribute = (CDMClassAttributes) cdmAttributesIn.elementAt(j);
       String attributeTag = tempClassAttribute.getTag();
       String attributeName = tempClassAttribute.getAttrName();
       String attributeType = mapAttributeType("OODB", tempClassAttribute);
       if (!(attributeTag == "FK" || attributeTag == "PKFK")) // the attribute is non-foreign key
          {
            ooClassIn.setOOattributes(attributeName, attributeType);
          }
     }
 }
```

Figure 8.4: The OODB class attribute rule

8.3.3 Components of the Data Convertor

The Data Convertor takes the CDM generated by the RDB Enricher and accesses the RDBMS to convert existing RDB data into the format defined by the target schemas. This module implements the algorithms and instance conversion rules described in Chapter 7. The Data Convertor consists of three main components as shown in Figure 8.1. These components include the OODB Data Generator (RDtoOOD), the ORDB Data Generator (RDtoORD) and the XML Data Generator (RDtoXML), which are responsible for converting RDB data into the corresponding OODB data, ORDB data and XML documents, respectively. For each of the three components, data stored as tuples in an RDB are converted into complex objects/literals in object-based databases or elements in XML documents. Firstly, the Convertor extracts tuples in RDB relations, and then transforms the extracted data to match the target format. Finally, the Convertor loads the transformed data into text files suitable for the target platform, in order to populate the schema translated by the Schema Translator. Each of the three modules has been implemented as a main Java class together with a set of methods corresponding to its instance conversion rules. However, further details are given here only for the OODB Data Convertor.

The OODB Data Convertor

The OODB Data Convertor, i.e., RDtoOOD component, generates the target data using a set of data instance conversion rules as described in Section 6.2. The RDtoOOD has been implemented as a main Java class that consists of eleven main methods representing the conversion rules. The RDtoOOD constructor invokes each of these methods when the corresponding rule needs to be applied. Sets of customised SQL queries are embedded in these methods to extract the desired data from an RDB. Once a query is executed, the result is transformed from the flat RDB form to the OODB format. The RDtoOOD generates the target data in script files as initial object files and relationship files. Firstly, the RDtoOOD generates a script file, for each concrete class, which contains OQL statements, i.e., OIF format, for instantiating classes by literal data types. Literal data types include atomic, multi-valued and composite attributes. Secondly, the RDtoOOD generates the files that contain OQL statements for establishing relationships among the objects defined earlier. Finally, the RDtoOOD generates codes added to the conversion program `MakeFile` in order to enact the generated data files. This process is performed based on the CDM structure, which facilitates the reallocation of attribute values in an RDB to the appropriate values or user-defined identifiers in the OODB.

Instantiating Classes: To define objects of a main class in the OODB, the RDtoOOD executes two SQL queries embedded in a method called `instantiateClass`. The first query extracts the primary key values from the RDB table corresponding to the OODB class being migrated. The second query extracts the non-foreign key values from the same RDB table. Then, from each tuple of these two queries, the RDtoOOD defines a user-defined object identifer, i.e., a surrogate OID by concatenating the CDM class name with the primary key values extracted from the corresponding RDB table. In addition, the RDB tuple comprising non-foreign key attributes is converted into the equivalent OQL statements, in order to define the object. Using the primary key values of the tuple, the RDtoOOD checks if the CDM class has any aggregation relationships. If so, the RDtoOOD invokes the `establishLitralAggregation` method that includes the suitable rules to perform this conversion. The `establishLitralAggregation` method takes the CDM class name and the aggregation relationship as arguments and returns the corresponding set of values or `struct` type. The generated OQL statements are stored in a file

named by the CDM class name, e.g., `Emp.oql`.

Instantiating Relationships: The defined objects are linked with other objects using foreign key values extracted from each RDB relation's tuples based on the relationships defined in the CDM class. This is the next step of conversion, establishing object-valued relationships as an implementation of the functions described in Section 7.2.2. Based on each CDM relationship, data are retrieved from the RDB table corresponding to the CDM class in which the relationship is defined, in order to instantiate the corresponding relationship defined in the OODB class mapped from the CDM class. The generated OQL statements are stored in a file for updating relationships, e.g., `Emp_relationships.oql`.

8.4 Summary

This chapter has described the prototype in which the concepts and algorithms of MIGROX solution have been implemented. Section 8.1 described the development environment and the reasons for choosing the software used in the prototype's development. Section 8.2 illustrated a high level design for the prototype. The three modules of the prototype and their main components are discussed. Section 8.3 presents in detail how each of these components has been implemented and how they communicate with each other.

The prototype provides a basis for a proof of concept. It shows that the concepts described in Chapters 4-7 can actually be implemented in terms of programming languages at a high level of automation. The prototype facilitates the migration process and, through its outputs, illustrates that MIGROX and its three phases can be practically executed. The current prototype can be considered as a basis for an integrated database migration tool.

The next chapter, Chapter 9, describes how an experimental study was conducted to evaluate MIGROX. The evaluation aims to check the equivalence between the input RDBs and each of the three target databases generated by the prototype.

Chapter 9

Evaluation of the Prototype

Chapter 8 has described the implementation of MIGROX prototype. The goal of this chapter is to evaluate the prototype, including a discussion of the experimental results. The evaluation aims at validating the solution presented in this dissertation by checking the equivalence between the input RDBs used in the database migration process and the three target databases generated by the prototype. Database equivalence checking includes the preservation of schema semantics, data, and integrity constraints. To the best of our knowledge, a comprehensive evaluation (i.e., one that takes all aspects of database equivalence) of a database migration solution has never been conducted before.

This chapter is structured as follows. Section 9.1 describes the criteria and methods used in the evaluation. Section 9.2 provides a description of an experimental environment, the hypotheses to be tested in the evaluation, and experimental setup. In Section 9.3, a detailed description and analysis of the experimental results is presented. Section 9.4 discusses the results obtained based on the efficiency of DBMSs in terms of RDB and ORDB query processing. Section 9.5 provides a summary of this chapter and points to what follows in the next chapter.

9.1 Evaluation Approach

Different cost metrics have been used in the literature to evaluate DBMS performance and in a limited way database migration processes. These metrics can be categorised as follows:

- **Schema information preservation**: Several user-based evaluation metrics have been proposed to verify the correctness of the schemas which resulted from schema mapping techniques. The generated target schema is compared to the input RDB schema to ensure that it reflects all the semantics of the input database. A schema is correct if all concepts of the underlying model are used correctly with respect to syntax and semantics [Fahrner and Vossen, 1995a]. In general, the result of an automatic database engineering process could be validated against the result that is obtained by an expert who performs the process manually [Chiang et al., 1996]. Target schemas can be evaluated syntactically by an expert user who is familiar with both source and target databases. Moreover, an automatically generated target schema can be compared with a schema translated from the same input database used in the existing literature [Lee et al., 2001]. Lee et al. [2001, 2003] show a proof of concept by comparing the DTD schema resulting from the implementation of the algorithms they proposed (i.e., FT and Net algorithms) with that of the DB2XML tool. DB2XML is a tool for transforming data from RDBs into XML [Turau, 1999].

- **Data equivalence**: Querying source and target databases can also be used to evaluate the migration results, e.g., validating the generated schemas. Fong and Cheung [2005] evaluate their approach by observing the difference between an RDB schema and its corresponding XML Schema according to query results provided by their software based on a sample of data loaded into the two databases. The same criterion has been used to observe the difference between an input RDB and its corresponding DTD results provided by both SQL Server for the RDB and Tamino XML Server [2009] for the DTD [Fong et al., 2006]. However, integrity constraints preservation has not been enough addressed in the current literature.

- **System efficiency comparison**: Most DBMS performance evaluations consider measurement of query elapsed times. The elapsed time metric is the amount of time query statements take to execute. A set of query-based benchmarks has been designed to test and measure different aspects of the functionality and performance of object-based and XML systems [Carey et al., 1993, 1997; Schmidt et al., 2001; Kurt and Atay, 2002; Runapongsa et al., 2006]. These benchmarks can be used to test the equivalence between source and target databases, resulting from database migration approaches. The OO1 [Cattell

and Skeen, 1992] and OO7 [Carey et al., 1993] benchmarks were designed to evaluate the performance of OODBMS. OO7 represents a comprehensive test of the wide range of OO features of OODBMS performance [Carey et al., 1993]. OO7 includes three sets of operations: traversals, queries and structural modifications. The BUCKY [Carey et al., 1997] and BORD [Lee et al., 2000] benchmarks were designed for ORDBMSs. BUCKY is a query-oriented benchmark which has been developed to test the maturity of an ORDB system's key features in relation to an RDB system [Carey et al., 1997]. The Michigan [Runapongsa et al., 2006] and XMark [Schmidt et al., 2001] benchmarks have been proposed for evaluating the performance of XML data management systems. XMark consists of an application scenario and a set of XQuery statements, which have been designed to assess the performance of XML query processors [Schmidt et al., 2001]. This benchmark tries to capture essential XML data processing which includes querying text search, hierarchical and ordered data. Kurt and Atay [2002] presented an experimental study of query processing efficiency of a native-XML database and an RDB. However, although query speed is not an essential issue for data migration methods, it is a very important evaluation issue for benchmarks that concern DBMS efficiency.

Our Approach: We evaluate our method in terms of the quality of the results produced by its prototype using experimental study. As we work with real databases, the experimental study is the most suitable approach to validate MIGROX and test the hypotheses. The evaluation is based on comparisons where a source RDB is compared with and measured against the target databases obtained by the prototype to check whether or not their schemas and data are equivalent. Factors for appraising equivalence of databases include preservation of semantics, data and integrity constraints. First, we conducted an experiment on databases used in the literature [Carey et al., 1997; Wang et al., 2005; Cattell and Barry, 2000; Elmasri and Navathe, 2006] to test the equivalence of source and target schemas. Second, we designed experiment based on the query-oriented BUCKY benchmark [Carey et al., 1997] to observe any differences between the source and target databases with regard to data content and integrity constraints. We used BUCKY and its queries as it is published, fully released and freely available benchmark. The BUCKY benchmark has been designed to test many of the key features offered by ORDB systems in relation to RDB systems. The

tested features include row types and inheritance, references and path expressions, sets of atomic values and references, and user-defined data types along with their methods [Carey et al., 1997]. The benchmark consists of an RDB and ORDB, including their semantically equivalent schemas, data and sets of queries. This experiment is implemented using a subset of the benchmark data. However, since our focus is to compare the quality of the target databases in a query-based experiment, some simplifications have been made to these queries without affecting the results of the comparison. We believe that a comprehensive evaluation can be achieved by these experiments.

Although we focus on assessing equivalence between source RDBs and the corresponding target databases to evaluate MIGROX, the system performance on data retrieval is also determined. We have used a subset of the queries used above to be run on the entire BUCKY database so as to appraise the relative efficiency of input and target databases based on query elapsed time.

9.2 Experimental Environment

In this section, a detailed description of the experiments is presented. The hypotheses and how the experiments were set up are explained in detail.

9.2.1 Hypotheses

The experiments conducted in this chapter aim to investigate the validity of the hypotheses that MIGROX is able to 1) preserve the structure and semantics of an existing RDB in a canonical data model (CDM); 2) generate equivalent target OODB, ORDB and XML schemas; and 3) effectively convert the RDB data into the target databases without redundancy or loss of data. The following criteria were investigated in our study with respect to the MIGROX evaluation:

- **The preservation of data semantics**: *Target schemas hold equivalent semantics of an existing RDB schema.* The CDM is able to preserve and enhance the RDB's integrity constraints and data semantics to fit in with the characteristics of the target databases. Generally, a translation process is called information-preserving if all possible database instances that can be represented

in the source schema can also be represented in the target database schema and vice versa [Fahrner and Vossen, 1995a].

- **The completeness of migration rules and data equivalence**: *Data instances of a source RDB are comparable to any of the three target databases generated by MIGROX.* The migration process is complete when all RDB data semantics that would be represented in target databases are accounted for in the schema translation and data conversion rules. This can be tested on real data stored in a database when all possible semantics expected to be generated from RDBs (e.g., inheritance, object-based relationships, multi-valued and composite attributes, etc) are considered to be covered in queries on the target databases and the data are retrieved without loss or redundancy.

9.2.2 Experimental Setup

Most existing studies have focused on measuring overall elapsed query times. In the experiments reported here, we appraise schema and data equivalence and compare query results from source and target databases. This is accomplished by observing the differences among the RDB and each of the target databases according to the results obtained. Two experiments have been designed to test the hypotheses using the criteria introduced in Section 9.2.1.

- Experiment I tests whether the target generated schemas have preserved the data semantics of the source database. Once a target schema is generated, it is compared to the input RDB schema to ensure that it reflects all the input RDB semantics. The CDM and target schema, generated from an existing RDB by the prototype, are correct when this target schema is equivalent to the schema mapped from the same RDB by existing manual approaches (i.e., [Carey et al., 1997; Urban and Dietrich, 2003; Elmasri and Navathe, 2006; Keivani, 2006]). The CDM is then validated as a representation of the existing RDB.

- Experiment II explores the equivalence of source and target data and integrity constraints based on the user-readable evaluation. This experiment is implemented on a small subset of the relational version of the university (UniDB) information system used in the BUCKY benchmark [Carey et al., 1997]. We have used the conceptual schema of the BUCKY benchmark and most of its

RDB and ORDB queries. We have made some modifications to BUCKY's logical schema and added some queries for testing integrity constraints. As the benchmark only provides the RDB and ORDB query versions, we have additionally translated the RDB queries into equivalent versions in an OODB and XML. Using query languages, the four sets of queries (i.e., for RDB, ORDB, OODB and XML) have been run on their respective DBMSs for retrieving information so that the user can check/observe/verify whether or not the results, which are small in size, are equivalent. Even though the aim of this experiment is to test the equivalence between source and target data and constraints, the query results could be interpreted for appraising database equivalence, including schema validation.

Database Descriptions

Source Databases: The main source RDB used in the experiments reported here is based on the BUCKY benchmark [Carey et al., 1997]. However, we have used other existing RDBs (i.e., UniDB [Carey et al., 1997], School [Urban and Dietrich, 2003] and Company [Elmasri and Navathe, 2006]) for the evaluation of semantic preservation that have been tested in Experiment I.

Figure 9.1 shows the conceptual schema of the UniDB in UML class diagram notation. The solid directed lines with a blank arrowhead represent generalisation relationships from more specialised classes, e.g., `Student` and `Employee` to more generalised classes, e.g., `Person`. In other words, `Student` and `Employee` are sub-classes of the superclass `Person`; `Staff` and `Instructor` are sub-classes of `Employee`; `Professor` is a sub-class of `Instructor`, and `TA` (teaching assistant) is a sub-class of both `Student` and `Instructor`. These generalisations form an inheritance hierarchy. The other lines represent associations between objects of the classes and are labelled on each side with a role name by which the association is known at that end and with multiplicities at the other end. For example, an object of the `Department` class must (i.e., 1..1, one and only one) have a *chair* who is in turn an object of the `Professor` class; inversely, an object of the `Professor` class may (i.e., 0..1, none or only one) *lead* a department. A multiplicity of 1..* means at least one and possibly many (e.g., a department can have one or many *employees* because 1..* is at the `Employee` end of the line and an employee *works in* one and only one department because 1..1 is shown at the `Department` end of the line).

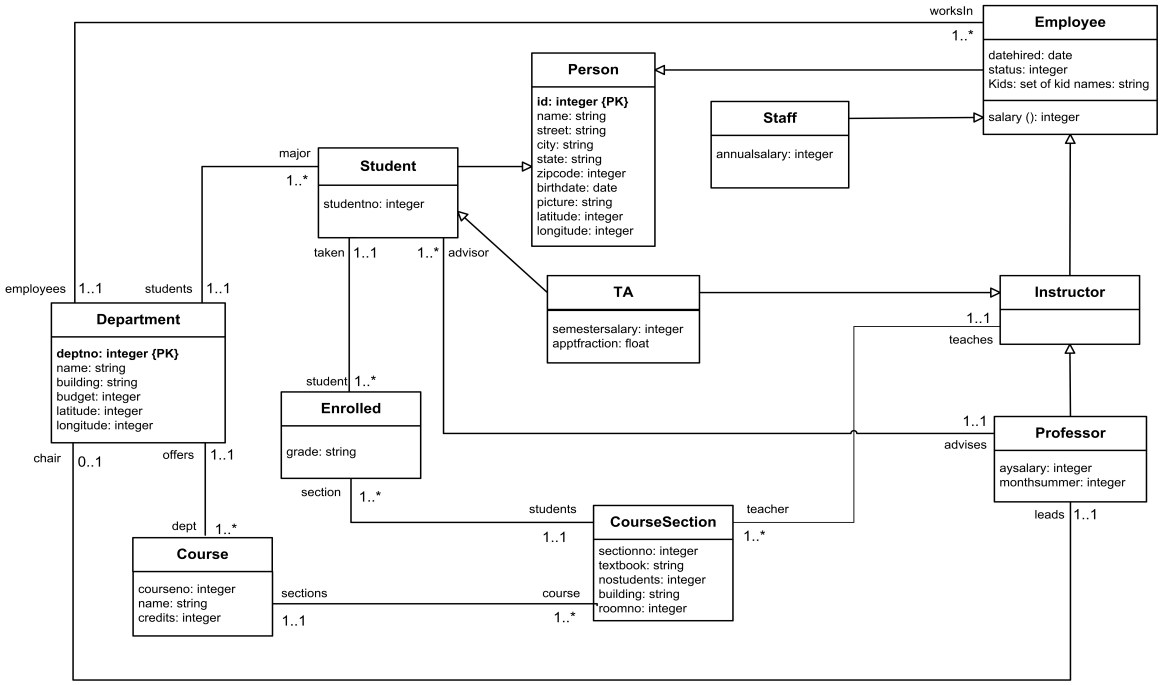

Figure 9.1: Conceptual schema for UniDB

Relational Implementation: Table 9.1 shows the logical schema of the UniDB. The basic relations in the implementation of the RDB version of the UniDB include: `Department`, `Person`, `Employee`, `Student`, `Staff`, `Instructor`, `TA`, `Professor`, `Course`, `CourseSection`, `Enrolled` and `Kids`. The relationships are modelled using primary/foreign key pairs. Primary keys are in **bold** font and foreign keys are in *italic*. Referential integrity is denoted as: foreign key → referenced relation. There are several alternative ways to model inheritance in relational data model [Elmasri and Navathe, 2006]. The approach implemented in BUCKY was to create a separate entity for each non-abstract type in the inheritance hierarchy, i.e., `Student`, `TA`, `Professor`, `Staff`. The common attributes are repeated in each table definition. Since there is no direct way to model inheritance relationship in the relational model, we create a separate relation for each super-class and for each of its sub-classes with their unique attributes and the primary key of the super-class. We assume this approach because it allows the extraction of inheritance relationships among tables automatically via the matching of primary/foreign keys without any user intervention. Given that multi-valued attributes are not supported in relational model, the

attribute *Kids* in the `Employee` class is modelled by creating an additional table called `Kids`, which has an *id* attribute corresponding to `Employee` and its sub-classes. Once the RDB schema is created, the data are bulk-loaded using the Oracle SQL*Loader. However, only a sample of RDB data and corresponding target data generated by MIGROX is used in Experiments I and II. Appendix B provides the DDL description of the RDB UinDB schema, the SQL*Loader control (CTL) files used to load its data, and the sample of RDB data.

Relation	Referential Integrity	Tuples
DEPARTMENT (**deptno**, name, building, budget, *chair*, latitude, longitude)	chair → PROFESSOR	250
COURSE (***deptno***, **courseno**, name, credits)	deptno → DEPARTMENT	12500
COURSESECTION((***deptno***, ***courseno***), **sectionno**, **semester**, *instructorid*, textbook, nostudents, building, roomno)	deptno, courseno→ COURSE, instructorid → INSTRUCTOR	50000
PERSON (**id**, name, street, city, state, zipcode, birthdate, picture, latitude, longitude)		125000
EMPLOYEE (***id***, *dept*, datehired, status)	id → PERSON, dept → DEPARTMENT	75000
INSTRUCTOR (***id***)	id → EMPLOYEE	50000
STAFF (***id***, annualsalary)	id → EMPLOYEE	25000
PROFESSOR (***id***, aysalary, monthsummer)	id → INSTRUCTOR	25000
STUDENT (***id***, studentno, *majordept*, *advisor*)	id → PERSON, majordept → DEPARTMENT, advisor → PROFESSOR	75000
TA (***id***, semestersalary, apptfraction)	id → INSTRUCTOR	25000
KIDS (***id***, **kidname**)	id → EMPLOYEE	116657
ENROLLED (***studentid***, (***deptno***, ***courseno***, ***sectionno***, ***semester***), grade)	studentid → STUDENT, deptno, courseno, sectionno, semester → COURSESECTION	150000

Table 9.1: Logical relational schema for the UniDB

Target Databases: Each target database equivalent to the UniDB is generated by the prototype as a set of files stored in a separate directory. Each directory contains a schema file, a set of files for object definitions, a set of files for relationship definitions, constraints files, and a program file that runs these files in a certain order to create each of the target databases. To load the database into the system, the program first enacts the schema file and then the files that contain object definitions in a first pass. Files containing primary keys, relationship definitions and constraints are loaded into the database in the second pass. However, the XML database directory contains only two files: one is an XML Schema document and the other is an XML instance

document. The physical schemas and fragments of data of the target databases generated by the prototype are provided in Appendix C.

Queries

To select appropriate queries for Experiments II, we consider the types of queries used to compare and contrast a wide range of data semantics converted from the RDB into each of the target databases, especially complex and user-defined types. Retrieval operations, including selecting, matching and joining are important in comparing the RDB semantics with those of the corresponding targets, e.g., association, aggregation and inheritance. Update operations such as `insert`, `delete` and `update` are taken into account in order to show that the integrity constraints in RDB are converted and preserved in the target databases. The criteria we have used in selecting the queries include: 1) queries should be simple and basic operations are supported in all database systems giving results that can be observed by the user; 2) all possible target database constructs should be covered; and 3) queries should focus on testing the equivalence between source and target databases, including semantics and data preservation. We used the queries of the BUCKY benchmark because they cover the fundamental areas to be tested, including inheritance hierarchies, object-based relationships, user-defined types and integrity constraints. In addition, the queries are designed to test the efficiency and speed of the data retrieval from the DBMSs. The following list gives the essential query types selected to be used in our experiments.

- **Selection**: Most of the results are obtained from databases using selection queries, including single and complex selections with relational operators.
- **Exact Match Lookup**: This type of query shows the ability of the database to handle simple lookups as a simple exact-match or over inheritance hierarchies.
- **Joins**: This type of query tests the ability of the database to perform joins, including single and inheritance joins.
- **Set Operations**: This type of query tests operations such as intersections and unions.
- **Set Membership**: This type of query tests for set membership, where the set is a collection of values extracted by selection statements.

- **Path-Expressions**: This type of query tests the ability to handle references to persistent objects. It specifies a path to related attributes and objects using the concept of a path expression. A path expression in object-based databases, which includes a navigation path through a relationship is similar to the outer join in RDBs.

- **User-defined Data Types**: This type of query is used for retrieving data stored as simple/composite multi-valued attributes and weak entities.

- **Data Manipulation Operations**: We have included this type of operation in Experiment II to evaluate the preservation of integrity constraints by updating database contents. Entity and referential integrities, unique keys as well as enforcing absence of null values are among the constraints to be tested. Insert, update and delete operations are covered in this category.

Testbed Configuration

The experiments reported in this chapter were conducted on a PC with Intel Pentium IV CPU clocked at 3.2 GHz, 1024 MB RAM and 80 GB of hard disk. The operating system used was Windows XP Professional. Oracle 11_g was employed for RDBs and ORDBs. The version of the OODBMS was lambda-DB 1.8 [Fegaras, 2008]. Lambda-DB is built on top of SHORE 1.1.1, operating under Red Hat Enterprise Linux Server release 5.3 (Tikanga). It supports ODMG ODL specifications and can handle most ODMG OQL queries [Fegaras, 2008]. Altova XMLSpy 2008 [Altova XMLSpy, 2008] was used for XML. Altova XMLSpy 2008 is an editor for creating, editing, and managing XML Schema and instance documents. It offers an XML editor and graphical schema designer along with a code generator, and it supports XPath and XQuery.

9.3 Experimental Results

This section reviews the results of our evaluations. We focus on two metrics, namely schema information preservation and data equivalence.

9.3.1 Experiment I: Testing Data Semantics Preservation

This experiment tests schema information preservation by comparing target schemas generated from the prototype with that translated from the same source schemas using existing manual schema mapping techniques (i.e., [Carey et al., 1997; Urban and Dietrich, 2003; Elmasri and Navathe, 2006; Keivani, 2006]). The evaluation includes comparisons of the schema structures, data semantics and integrity constraints.

OODB schemas

Urban and Dietrich used UML features to illustrate mapping alternatives from UML to relational, object-oriented and object-relational data models [Urban and Dietrich, 2003]. The approach used an RDB called School to compare and contrast mapping techniques specific to each model. Figure 9.2 shows the RDB logical School schema they used.

Person (**pID**, dob, firstName, lastName)
Student (***pID***, status, *major*): pID → Person, major → Department
Faculty (***pID***, rank, *dept*): pID → Person, dept → Department
Department (**code**, name, *chair*): chair → Faculty
CampusClub (**cID**, name, phone, location, *advisor*): advisor → Faculty
Clubs(***pID***, ***cID***): pID → Student, cID → CampusClub

Figure 9.2: Relational schema of School database [Urban and Dietrich, 2003]

Figure 9.3 shows a fragment of the ODMG 3.0 ODL schema mapped from Urban and Dietrich [2003] and Figure 9.4 shows the equivalent schema generated by MIGROX. A full description of both schemas is provided in Appendix D. In the figures it can be observed that apart from methods associated with classes, which are not assumed in our approach, the results of MIGROX and Urban and Dietrich's approach were similar in translating the RDB schema into its equivalent OODB schema, including classes, attributes, single/collection-based relationships, inheritance, and keys. However, association relationships are mapped in MIGROX bi-directionally and aggregation relationships uni-directionally. Such relationships were modelled as bi-directional and sometimes uni-directional relationships in Urban and Dietrich's work, e.g., the *deptChair* attribute defined in the `Department` class. Another example is that the *major* association relationship in `Student` class is translated by MIGROX bi-directionally, with its inverse the *students* relationship in the `Department` class. Such a relationship is mapped by Urban and Dietrich as *major* attribute in

the `Student` class whose type is a single instance of the `Department` class, and the *students* attribute in the `Department` class was defined as a collection of students.

```
class Student extends Person(extent students){
attribute string status; attribute Department major;
relationship set<CampusClub> memberOf inverse CampusClub::members;
.  .  .  }
class Department(extent departments key code) {
attribute string code; attribute string name; attribute Faculty deptChair; attribute set<Student>
students; attribute set<Faculty> deptFaculty;
.  .  .  }
class CampusClub(extent campusClubs key cID){
attribute string cID; attribute string name; attribute string location; attribute string phone;
relationship set<Student> members inverse Student::memberOf;
relationship Faculty advisor inverse Faculty::advisorOf;
.  .  .  }
```

Figure 9.3: Fragment of OODB School schema mapped from Urban and Dietrich [2003]

```
class Student extends Person (extent Students) {
attribute string status;
relationship set<Campusclub> memberof inverse Campusclub::members;
relationship Department major inverse Department::students
};
class Department (extent Departments key code) {
attribute string code; attribute string name;
relationship set<Faculty> deptfuclty inverse Faculty::dept;
relationship set<Student> students inverse Student::major;
relationship Faculty deptchair inverse Faculty::chairof;
};
class Campusclub (extent Campusclubs key cid) {
attribute string cid; attribute string name; attribute string phone; attribute string location;
relationship set<Student> members inverse Student::memberof;
relationship Faculty advisor inverse Faculty::advisorof;
};
```

Figure 9.4: Fragment of OODB School schema generated by the MIGROX prototype

ORDB schemas

Several RDB schemas have been translated into corresponding ORDB schemas [Carey et al., 1997; Urban and Dietrich, 2003; Keivani, 2006]. Figure 9.5 shows a fragment of the ORDB SQL3 schema mapped by Urban and Dietrich approach and Figure 9.6 shows the equivalent schema generated by MIGROX. Further descriptions of both schemas can be found in Appendix D.

Considering the differences in SQL syntax between Oracle (the result of MIGROX) and the SQL3 standard, the same schemas, including relations and constraints are translated by MIGROX and the above approaches into equivalent ORDB object types, including attributes, references among objects and nested tables/arrays, and object

```
create type student_udt under person_udt as (
status varchar(20),
clubs ref(campusclub_udt) scope campusclub array[20],
major ref(department_udt) scope department) not final
method getclubs() returns varchar(25) array[20];

create type campusclub_udt as (
cid varchar(11), name varchar(25), location varchar(25), phone varchar(25),
advisor ref(faculty_udt) scope faculty,
members ref(student_udt) scope student array[100]) not final
ref is system generated;

create type department_udt as (
code varchar(3), name varchar(40),
deptchair ref(faculty_udt) scope faculty) not final
ref is system generated
method getstudents() returns varchar(40) array[1000],
method getfaculty() returns varchar(40) array[50];

create table student of student_udt under person;
create table department of department_udt (
constraint department_pk primary key(code),
ref is oid system generated);
create table campusclub of campusclub_udt (
constraint campusclub_pk primary key(cid),
ref is oid system generated);
```

Figure 9.5: Fragment of ORDB School schema mapped from Urban and Dietrich [2003]

```
create or replace type student_t under person_t (
status char(10), clubs campusclub_ntt, major ref department_t) final;
/
create or replace type campusclub_t as object (
cid char(10), name char(20), phone char(10), location char(30),
members student_ntt, advisor ref faculty_t) not final;
/
create or replace type department_t as object (
code char(3), name char(20), faculties faculty_ntt, students student_ntt, deptchair ref faculty_t)
not final;
/
create table campusclub of campusclub_t
nested table members store as members_nt
;
create table department of department_t
nested table faculties store as faculties_nt
nested table students store as students_nt
;
create table student of student_t
nested table clubs store as clubs_nt
;
alter table campusclub add constraint campusclub_pk primary key (cid);
alter table department add constraint department_pk primary key (code);
alter table student add (scope for (major) is department);
alter table campusclub add (scope for (advisor) is faculty);
...
```

Figure 9.6: Fragment of ORDB School schema generated by the MIGROX prototype

tables. Similar to MIGROX, object types, which may be defined as inheritance hierarchies have been used to create object tables [Carey et al., 1997; Keivani, 2006; Urban and Dietrich, 2003], and primary keys are defined for those tables. Unlike in the other approaches, RDB relationships are translated by MIGROX into object-based relationships bi-directionally. The 1 side of relationships is translated as `ref`. However, as arrays have their drawbacks, such as fixed size, MIGROX is similar to Keivani [2006] in mapping the M side of relationships as nested tables. Nested tables are supported by Oracle and are designed to handle unordered and unlimited elements. Nested tables are especially appropriate for modelling association/aggregation collection data types. Urban and Dietrich [2003] used an SQL3 array of `ref`s to map the multi-valued relationships. Unlike other approaches, they proposed using triggers for referential integrity maintenance in the ORDB schema [Urban and Dietrich, 2003].

XML Schemas

Elmasri and Navathe [2006] described a general algorithm for mapping an EER into an RDB schema and then into XML Schema. A database called Company is used to illustrate the mapping steps (see Chapters 7 and 27 for ER diagram, and the corresponding RBD schema and the XML schema document [Elmasri and Navathe, 2006]). We have used the Company RDB as input for the MIGROX prototype, aiming to generate an XML document from it. The XML Schema file generated by MIGROX from this database is comparable to the XML Schema file mapped from Elmasri and Navathe [2006], a fragment of which is shown in Figure 9.7. The equivalent schema generated by the MIGROX prototype is given in Figure 9.8. The two XML Schema documents generated by both approaches can be found in Appendix E.

The description of both schema documents, including namespaces, first level elements and their types, minimum and maximum occurrences, key specifications and composed attributes, are specified similarly in both approaches. Elements are specified with a type attribute so that the structure of the elements are defined separately. Complex types are defined with sequences of sub-elements corresponding to the attributes of RDB relations. Multi-valued attributes are specified with `maxOccurs` = "`unbounded`" in the corresponding element. Generally, in terms of semantics and information preservations, it was found that both XML Schema documents were comparable. However, MIGROX maps more precisely the attributes and their types and whether each attribute is optional or required.

```
<xsd:complexType name = "Employee">
   <xsd:sequence>
     <xsd:element name = "employeeName" type= "Name"/>
     <xsd:element name = "employeeSsn" type= "xsd:string"/>
     <xsd:element name = "employeeSex" type= "xsd:string"/>
     <xsd:element name = "employeeSalary" type= "xsd:unsignedInt"/>
     <xsd:element name = "employeeBirthdate" type= "xsd:date"/>
     <xsd:element name = "employeeDepartmentNumber" type= "xsd:string"/>
     <xsd:element name = "employeeSuperSSN" type= "xsd:string/>
     <xsd:element name = "employeeAddress" type= "Address"/>
     <xsd:element name = "employeeWorksOn" type= "WorksOn" minOccurs= "1" maxOccurs=
"unbounded"/>
     <xsd:element name = "employeeDependent" type= "Dependent" minOccurs= "0" maxOccurs=
"unbounded"/>
   </xsd:sequence>
</xsd:complexType>
...
<xs:complexType name = "Name">
   <xs:sequence>
     <xs:element name = "firstName" type= "xs:string"/>
     <xs:element name = "middleName" type= "xs:string"/>
     <xs:element name = "lastName" type= "xs:string"/>
   </xs:sequence>
</xs:complexType>
...
<xs:complexType name = "Dependent">
   <xs:sequence>
     <xs:element name = "dependentName" type= "xs:string"/>
     <xs:element name = "dependentSsex" type= "xs:string"/>
     <xs:element name = "dependentBirthDate" type= "xs:date"/>
     <xs:element name = "dependentRelationship" type= "xs:string"/>
   </xs:sequence>
</xs:complexType>
```

Figure 9.7: Fragment of XML Company schema mapped from Elmasri and Navathe [2006]

```
<xs:complexType name = "Employee_t">
   <xs:sequence>
     <xs:element name = "fname" type= "xs:string"/>
     <xs:element name = "minit" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "lname" type= "xs:string"/>
     <xs:element name = "ssn" type= "xs:int"/>
     <xs:element name = "bdate" type= "xs:date" minOccurs= "0"/>
     <xs:element name = "address" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "sex" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "salary" type= "xs:int" minOccurs= "0"/>
     <xs:element name = "superssn" type= "xs:int" minOccurs= "0"/>
     <xs:element name = "dno" type= "xs:int"/>
     <xs:element name = "hasDependent" type= "Dependent_t" minOccurs= "0" maxOccurs=
"unbounded"/>
   </xs:sequence>
</xs:complexType>
...
<xs:complexType name = "Dependent_t">
   <xs:sequence>
     <xs:element name = "dependent_name" type= "xs:string"/>
     <xs:element name = "sex" type= "xs:string"/>
     <xs:element name = "bdate" type= "xs:date" minOccurs= "0"/>
     <xs:element name = "relationship" type= "xs:string" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
```

Figure 9.8: Fragment of XML Company schema generated by the MIGROX prototype

Unlike MIGROX, composite attributes such as `Name` and `Address` are specified as complex types and embedded directly in their parent elements by Elmasri and Navathe [2006]. The *fname*, *minit* and *lname* attributes that represent the name of an `Employee` are, conceptually, a composite attribute, and is normally identified at the conceptual design stage. So the fact that the `Name` is composite in XML is because of it is being composite in the ER model (see Elmasri and Navathe [2006], page 218). However, it has been broken down into atomic parts in the relational representation. There is no way to identify this composition automatically from the RBD schema in Elmasri and Navathe [2006] without human input. However, although such a mapping uses the features of XML Schema language, it seems that from a design perspective it is meaningless to map these attributes as separate types from the RDB, as each employee has one address and one name. MIGROX maps these attributes in a flat relational format. However, if these attributes are represented in separate relations with the key of the `Employee` relation, then MIGROX will map them in a similar way to Elmasri and Navathe [2006]. As these attributes are part of the `Employee` relation, MIGROX maps them as sub-elements of the `Employee` complex type.

The `Works_on` relation is translated into two complex types embedded into their parent elements as compound attributes by Elmasri and Navathe [2006], whereas it is mapped by MIGROX as main element under the schema.

In summary, in this experiment we have compared MIGROX with existing manual mapping techniques. These techniques give the user an opportunity to use all features of target models and their conceptual schemas, resulting in well-designed physical schemas. By comparing the results of these techniques with the results of the MIGROX prototype, we found that both sets of results were comparable. The resulting schemas show MIGROX's algorithms and existing manual algorithms to be equivalence-preserving translations. Furthermore, MIGROX is a fully-automatic approach and has the ability to generate more accurate and correct target schemas. In addition, MIGROX was more comprehensive than its manual counterparts because it generates (if desired by the user) one target database (or its schema only) or up to three different databases. Therefore, the CDM, which preserves an enhanced structure of an existing RDB, is translatable into any of the three target database schemas. The algorithms of CDM generation and schema translation are correct in the sense that they have all preserved the original information of the RDBs. Many semantics

have been converted from an RDB into the target databases, e.g., association, aggregation and inheritance. Moreover, the main type of constraints that can be extracted from an RDB, including key constraints, constraints on NULLs and entity and referential integrity constraints, are all translated explicitly into the equivalent target schemas.

9.3.2 Experiment II: Testing Data Equivalence and the Completeness of Migration Rules

This experiment explores the equivalence between source and target databases in terms of data and integrity constraints. The experiment involves a set of queries applied on a subset of the RDB version of the UniDB and each of the three target databases generated by the MIGROX prototype. This section presents the results of the experiment, which are measured based on user readable metrics. The description of each query, its code in RDB SQL and its equivalents in OQL, ORDB SQL and XQuery as well as the query results, are given. We explain the role of each query and any observations concerning the results if they vary.

Query 1: SINGLE-EXACT

Query	*Find the name, building and budget of the department with number 1.*
RDB	`select name, building, budget from department where deptno = 1;`
ORDB	`select name, building, budget from department where deptno = 1;`
OQL	`select struct(name:d.name, building:d.building, budget:d.budget) from d in Departments where d.deptno = 1`
XQuery	`for $s in doc("XMLSchema.xml")//department where $s/deptno = 1 return <res> { $s/name/text(), $s/ building/text(), $s/budget/text()} </res>`

Result	name	building	budget
	`deptname1`	`building`	`6000000`

This query retrieves data from one type as a simple exact-match lookup. The query returns details of department 1. `Departments` in the OODB query is the extent of the `Department` class, i.e., an entry point to return objects from it as a collection of `struct`. The `text()` in the XML query gives the data of the attributes without the enclosing tags.

Query 2: HIER-EXACT - Exact Match Over Inheritance Hierarchy

Query	*Find the name and annual salary of the staff with id 2.*
RDB	`select p.name, s.annualsalary from person p, staff s where s.id = p.id and s.id = 2;`
ORDB	`select name, annualsalary from staff where id = 2;`
OQL	`select struct(name:s.name, salary:s.annualsalary) from s in Staffs where s.id = 2`
XQuery	`for $x in doc("XMLSchema.xml")//staff where $x/id = 2 return <res> {$x/name/text(), $x/annualsalary/text()} </res>`

Result	name	annualsalary
	staffName2	33000

This query returns data over the inheritance hierarchy. The join operation is required in the RDB to join the `Staff` sub-table with its super-table `Person` to retrieve the desired data over the hierarchy, however it is not needed in the target database queries.

Query 3: SINGLE-METH - Method Query Over One Type

Query	*Find IDs of all Professors who make 145000 or more per year.*
RDB	`select id from professor p where (p.aysalary * (9 + p.monthsummer)/ 9.0) >= 145000;`
ORDB	`select id from professor p where (p.aysalary * (9 + p.monthsummer)/ 9.0) >= 145000;`
OQL	`select p.id from p in Professors where (p.aysalary * (9 + p.monthsummer)/ 9.0) >= 145000`
XQuery	`for $x in doc("XMLSchema.xml")//professor where ($x/aysalary * ( 9 + $x/monthsummer) div 9.0) >= 145000 return <res> {$x/id/text()}</res>`

Result	id
	65805
	54453

This query returns salary calculations over one table/class. For object-based systems, this calculation could be done by invoking methods, which are described in Section 9.4 as part of query performance testing.

Query 4: HIER-METH - Method Query over Inheritance Hierarchy

Query *Find names and addresses of all Employees who make 140000 or more per year.*

RDB
```
select p.name, p.street, p.city, p.zipcode from person p,
staff s where p.id = s.id and s.annualsalary >= 140000 union
select p.name, p.street, p.city, p.zipcode from person
p, professor f where p.id = f.id and (f.aysalary
* (9 + f.monthsummer) / 9.0) >= 140000 union
select p.name, p.street, p.city, p.zipcode from person p, ta t where
p.id = t.id and apptfraction * (2 * t.semestersalary) >= 140000;
```

ORDB
```
select s.name, s.street, s.city, s.zipcode from
staff s where s.annualsalary >= 140000 union
select p.name, p.street, p.city, p.zipcode from professor p
where (p.aysalary * (9 + p.monthsummer) / 9.0) >= 140000 union
select t.name, t.street, t.city, t.zipcode from ta t where apptfraction
* (2 * t.semestersalary) >= 140000;
```

OQL
```
select struct(name:s.name, street:s.street, city:s.city, zipcode:
s.zipcode) from s in Staffs where s.annualsalary >= 140000 union
select struct(name:p.name, street:p.street, city:p.city,
zipcode:  p.zipcode from p in Professors where
(p.aysalary * (9 + p.monthsummer) / 9.0) >= 140000 union
select struct(name:t.name, street:t.street, city:t.city, zipcode:
t.zipcode from t in Tas where t.apptfraction * (2 * t.semestersalary)
>= 140000
```

XQuery
```
for $s in doc("XMLSchema.xml")//staff where $s/annualsalary
>= 140000 return <res> { $s/name/text(), $s/street/text(),
$s/city/text(), $s/zipcode/text() } </res>,
for $p in doc("XMLSchema.xml")//professor where ($p/aysalary * (9 +
$p/monthsummer) div 9.0) >=140000 return <res> { $p/name/text(),
$p/street/text(), $p/city/text(), $p/zipcode/text() } </res>,
for $t in doc("XMLSchema.xml")//ta where $t/apptfraction * (2
* $t/semestersalary) >= 140000 return <res> { $t/name/text(),
$t/street/text(), $t/city/text(), $t/zipcode/text()}</res>
```

Result	name	street	city	zipcode
	professorName47254	xxxx_streetname	city_name	34306
	professorName54453	xxxx_streetname	city_name	23403
	professorName59247	xxxx_streetname	city_name	98075
	professorName65805	xxxx_streetname	city_name	92344

This query returns salary calculations over the inheritance hierarchy. As the `Employee` type is in the middle of the hierarchy, searching its sub-classes using the union operation is required in object-based/XML queries. In the RDB, this query is similar considering the top-level super-class, i.e., `Person` table to be joined with all other

sub-classes involved in the hierarchy.

Query 5: SINGLE-JOIN - Relational Join Query

Query *Find the names, buildings and budgets of all departments with the same budget*

RDB
```
select d1.name, d1.building, d1.budget, d2.name, d2.building, d2.budget
from department d1, department d2 where d1.budget = d2.budget and
d1.deptno < d2.deptno;
```

ORDB
```
select d1.name, d1.building, d1.budget, d2.name, d2.building, d2.budget
from department d1, department d2 where d1.budget = d2.budget and
d1.deptno < d2.deptno;
```

OQL
```
select struct(name1:d1.name, building1:d1.building, budget1:d1.budget,
name2:d2.name, building2:d2.building, budget2:d2.budget) from d1
in Departments, d2 in Departments where d1.budget = d2.budget and
d1.deptno < d2.deptno
```

XQuery
```
for $x in doc("XMLSchema.xml")//department, $y in doc("XMLSchema.xml")
//department where $x/budget = $y/budget and $x/deptno < $y/deptno
return <res> { $x/name/text(), $x/building/text(), $x/budget/text(),
$y/name/text(), $y/building/text(), $y/budget/text() } </res>
```

Result	name1	building1	budget1	name2	building2	budget2
	deptname1	building	6000000	deptname220	building	6000000

This query retrieves data using self-join processing. As such join operations can be specified in all the query languages used, the structures of all versions of the query were very close.

Query 6: HIER-JOIN - Relational Join Over Inheritance Hierarchy

Query *Find all TAs with the same hired date as those live in the same zip code area.*

RDB
```
select p1.id, p1.name, p2.id, p2.name from person p1, person
p2, employee e1, employee e2, ta t1, ta t2 where e1.datehired =
e2.datehired and p1.zipcode = p2.zipcode and p1.id < p2.id and p1.id =
e1.id and p2.id = e2.id and p1.id = t1.id and p2.id = t2.id;
```

ORDB
```
select t1.id, t1.name, t2.id, t2.name from ta t1, ta t2 where
t1.datehired = t2.datehired and t1.zipcode = t2.zipcode and t1.id <
t2.id;
```

OQL
```
select struct(id1:t1.id, name1:t1.name, id2:t2.id, name2:t2.name) from
t1 in Tas, t2 in Tas where t1.datehired = t2.datehired and t1.zipcode =
t2.zipcode and t1.id < t2.id
```

XQuery
```
for $x in doc("XMLSchema.xml")//ta, $y in doc("XMLSchema.xml")//ta
where $x/datehired = $y/datehired and $x/zipcode = $y/zipcode and $x/id
< $y/id return <res>{$x/id/text(), $x/name/text(), $y/id/text(),
$y/name/text() } </res>
```

Result	id1	name1	id2	name2
	47397	studentName47397	71661	studentName71661

This query uses explicit joins between types in inheritance hierarchies. Unlike target database queries, all top-level super-tables, i.e., `Person` and `Employee` of the sub-table `TA`, are to be accounted for in the join operation in the RDB version of the query.

Query 7: SET-ELEMENT - Set Membership

Query *Find ids, names and addresses of all staff who have a child named boy90.*

RDB
```
select p.id, p.name, p.street, p.city, p.state, p.zipcode from person
p, staff s, kids k where p.id = k.id and s.id = k.id and k.kidname =
'boy90';
```

ORDB
```
select s.id, s.name, s.street, s.city, s.state, s.zipcode from staff s,
table (s.kidnames) k where k.kidname = 'boy90';
```

OQL
```
select struct(id:s.id, name:s.name, street:s.street, city:s.city,
state:s.state, zipcode:zipcode) from s in Staffs where "boy90" in
s.kidnames
```

XQuery
```
for $x in doc("XMLSchema.xml")//staff where $x/kidnames = 'boy90'
return <res> { $x/id/text(), $x/name/text(), $x/street/text(),
$x/city/text(), $x/state/text(), $x/zipcode/text()} </res>
```

Result	id	name	street	city	state	zipcode
	1	staffName1	xxxx_streetname	city_name	Oklahoma	41421
	3	staffName3	xxxx_streetname	city_name	Florida	18456

This query shows how to handle collection-valued attributes. The RDB query involves a join with the `Kids` table, whereas a set of strings, a nested table and a sub-element containing the collection-valued fields are used in the OODB, ORDB and XML query versions, respectively.

Query 8: SET-AND - Anded Set Membership

Query *Find ids, names and addresses of all Staff who have children named girl90 and boy90.*

RDB
```
select p.id, p.name, p.street, p.city, p.state, p.zipcode from person
p, staff e, kids k1, kids k2 where e.id = p.id and e.id = k1.id and
e.id=k2.id and k1.kidname = 'girl90' and k2.kidname = 'boy90';
```

ORDB
```
select s.id, s.name, s.street, s.city, s.state, s.zipcode from staff
s, table (s.kidnames) k1, table (s.kidnames) k2 where k1.kidname =
'girl90' and k2.kidname = 'boy90';
```

OQL
```
select struct(id:s.id, name:s.name, street:s.street, city:s.city,
state:s.state, zipcode:zipcode) from s in Staffs where "girl90" in
s.kidnames and "boy90" in s.kidnames
```

XQuery
```
for $x in doc("XMLSchema.xml")//staff let $m := $x/kidnames, $n
:= $x/kidnames where $m = 'girl90' and $n = 'boy90' return <res>
{$x/id/text(), $x/name /text(), $x/street /text(), $x/city /text(),
$x/state /text(), $x/zipcode/text()} </res>
```

Result	id	name	street	city	state	zipcode
	1	staffName1	xxxx_streetname	city_name	Oklahoma	41421

This query is similar to the SET-ELEMENT query used for finding data in collection types, however, it is slightly more complex.

Query 9: 1HOP-NONE - Single-Hop Path, No Selection

Query *Find the details of all student/major pairs.*

RDB
```
select p.id, p.name, p.state, d.deptno, d.name from person p,
department d, student s where p.id = s.id and s.majordept = d.deptno;
```

ORDB
```
select s.id, s.name, s.state, s.major.deptno, s.major.name from student
s;
```

OQL
```
select struct(id:s.id, name:s.name, state:s.state,
deptno:s.major.deptno, deptname:s.major.name) from s in Students
```

XQuery
```
for $x in doc("XMLSchema.xml")//student, $y in doc("XMLSchema.xml")//
department where $x/majordept = $y/deptno return <res> {
$x/id/text(), $x/name/text(), $x/state/text(), $y/deptno/text(),
$y/name/text()}</res>
```

Result	id	name	state	deptno	deptname
	65990	studentName65990	Vermont	1	deptname1
	75001	studentName75001	Georgia	1	deptname1
	75051	studentName75051	Florida	10	deptname10
	117525	studentName117525	Oklahoma	10	deptname10
	108981	studentName108981	West_Virginia	220	deptname220
	47397	studentName47397	Iowa	25	deptname25
	71661	studentName71661	Idaho	25	deptname25

This query finds data using path expressions. In object-based databases, references are de-referenced using the "." symbol in path expressions. References hide the join operations, which are explicitly necessary for this kind of query in an RDB and XML. In the RDB query, the top level super-table, i.e., `Person` is included in the join. References simplify query writing considerably.

Query 10: 1HOP-ONE - Single-Hop Path, One-Side Selection

Query	*Find the major of the student named studentName75001.*
RDB	`select p.id, p.name, d.deptno, d.name, d.building from person p, student s, department d where p.id = s.id and s.majordept = d.deptno and p.name= 'studentName75001';`
ORDB	`select s.id, s.name, s.major.deptno, s.major.name, s.major.building from student s where name= 'studentName75001';`
OQL	`select struct(id:s.id, name:s.name, deptno:s.major.deptno, deptname:s.major.name, building:s.major.building) from s in Students where s.name= "studentName75001"`
XQuery	`for $x in doc("XMLSchema.xml")//student, $y in doc("XMLSchema.xml")// department where $x/majordept = $y/deptno and $x/name= 'studentName75001' return <res> {$x/id/text(), $x/name/text(), $y/deptno/text(), $y/name/text(), $y/building/text() } </res>`

Result	id	name	deptno	deptname	building
	75001	studentName75001	1	deptname1	building

This query is an exact-match lookup (using path expressions) of a department starting from the `Student` type. In object-based databases, data can also be retrieved

using the opposite direction of the relationship since we define the relationship bi-directionally. Starting from departments and following the set references to the students can return the same data. To extract the same data, joins are again required in the RDB and XML query versions.

Query 11: 1HOP-MANY - Single-Hop Path, Many-Side Selection

Query	*Find ids and names of all students majoring in Department1.*
RDB	`select p.id, p.name from person p, student s, department d where p.id = s.id and s.majordept = d.deptno and d.name = 'deptname1';`
ORDB	`select st.column_value.id, st.column_value.name from department d, table(d.students) st where d.name = 'deptname1';`
OQL	`select struct(id:s.id, name:s.name) from d in Departments, s in d.students where d.name = "deptname1"`
XQuery	`for $x in doc("XMLSchema.xml")//student, $y in doc("XMLSchema.xml")//department where $x/majordept = $y/deptno and $y/name= 'deptname1' return <res>{$x/id/text(), $x/name/text()}</res>`

Result	id	name
	65990	studentName65990
	75001	studentName75001

Joining the related types, this query retrieves the same data from the RDB and XML systems. In object-based query versions, starting from departments, the queries follow the path to the set of references to the students to return the data. The `column_value` is used in the ORDB query to return the objects from nested tables whose tuples are actually references to tuples in other tables.

Query 12: 2HOP-ONE - Two-Hop Path, Many-Side Selection

Query	*Find the semester, enrolment limit, department number, and department name for all sections of courses taught in room 50.*
RDB	`select x.semester, x.nostudents, d.deptno, d.name from coursesection x, course c, department d where x.deptno = c.deptno and x.courseno = c.courseno and c.deptno = d.deptno and x.roomno = 50;`
ORDB	`select se.column_value.semester, se.column_value.nostudents, d.deptno, d.name from department d, table(d.offers) co, table(co.column_value.sections) se where se.column_value.roomno = 50;`

OQL
```
select struct(semester:s.semester, nostudents:s.nostudents,
deptno:d.deptno, deptname:d.name) from d in Departments, o in d.offers,
s in o.section where s.roomno = 50
```

XQuery
```
for $x in doc("XMLSchema.xml")//department, $y in doc("XMLSchema.xml")
//course, $z in doc("XMLSchema.xml") //coursesection where $x/deptno =
$y/deptno and $y/courseno = $z/courseno and $y/deptno = $z/deptno and
$z/roomno = 50 return <res> {$z/semester/text(), $z/nostudents/text(),
$x/deptno/text(), $x/name/text()} </res>
```

Result	semester	nostudents	deptno	deptname
	1	20	220	deptname220
	2	20	220	deptname220

This query retrieves data with longer paths and more complex collection types. The desired data are obtained from the RDB and XML by joining the `Department`, `Course` and the `Coursesection` tables/elements. The same data are returned in the ORDB using two hop paths, starting from the `Department` through the path of nested table `offers`, which is a collection of references to the `Courses` and then the nested table `sections` defined in the `Course`, which is a collection of references to the `Coursesection`. This query shows the preservation of a complex ref-based aggregation relationship, as each department offers a collection of courses (as components) and similarly each course consists of a collection of sections.

Query 13: Checking primary key constraints

Query *Insert data of a new depatment with number 25.*

RDB
```
insert into department values (25, 'deptname25', 'building', 8000000,
47254, 55, 12);
```

ORDB
```
insert into department values (25, 'deptname25', 'building', 8000000,
55, 12, course_ntt(), null, null, null);
```

OQL
```
%d25 := persistent Department (deptno:  25, name:  "deptname25",
building:  "building", budget:  8000000, latitude:  55, longitude:
12);
```

XQuery
```
the XML document has been edited manually by adding a department
elememt instance with number 25
```

Result
```
unique constraint violated
```

This query is to ensure that primary key constraints defined in the source schema are translated into the target schemas. We need to check that no two objects in

one table/class/element can have the same value for the primary key. Every primary key, which must be unique can be made up of a single attribute or combination of attributes. For example, duplication in *deptno* gives an error as the primary key constraint is violated. That is, every department object has a number, and that number is distinct from every other department objects. For each database, we inserted a tuple that contains a value of an attribute that is specified as a primary key and is already inserted. The tuple is rejected by all the DBMSs.

Query 14: Checking unique and null/not null constraints

Query	*Insert data of a new depatment with name deptname25.*
RDB	`insert into department values (26, 'deptname25', 'building', 8000000, 47254, 55, 12);`
ORDB	`insert into department values (26, 'deptname25', 'building', 8000000, 55, 12, course_ntt(), null, null, null);`
OQL	`unique, null and not null constraints are not supported`
XQuery	`the current XML document has been edited manually by adding a department elememt with name deptname25`
Result	`unique constraint violated`

Several queries have been designed to ensure that null/not null and unique constraints specified in the source schema are translated into the target schemas, if applicable. In these queries we have ensured that no two objects can have the same value for the unique key, and that every object must contain a value for the attribute that is specified as not null. For each database (apart from the OODB), we have inserted tuples to test these constraints, and found that the tuples have been rejected by the DBMSs. As the attribute *name* is restricted to be unique, the department name `department25`, which is already stored in a department with number 25, was rejected. Similarly, we cannot insert null into *name* attribute in a department as it is specified in the target schema as not null. The `unique` keyword is similar to the `key` in XML. The names for all departments must be unique. Nullable/not nullable fields in elements, which are represented with `minOccurs`="0" where the field is nullable, or `minOccurs`="1" where the field is not nullable, are validated. The test shows that these constraints are also preserved in the XML Schema document. For example, a null value inserted into the *name* field in an instance of the `Department` element has been rejected. This is because the *name* attribute was specified with `minOccurs`="1".

Query 15: Checking referential integrity constraints

Query	*Delete the professor identified by id = 59247.*
RDB	`delete from professor where id = 59247;`
ORDB	`delete from professor where id = 59247;`
OQL	`x:= element (select * from p in Professors where id = 59247` `x → delete()`
XQuery	`the professor element identifed by id = 59247 has been removed from the XML document manually`
Result	`referintial integrity constraint violated`

Referential integrity in RDB ensures that relationships between tables remain consistent. Referential integrity also includes the cascading update and delete, which ensure that changes made to the foreign key table are reflected in the primary key table (cascading techniques are not covered in our method). This means that, if a table has a foreign key referencing another table, we cannot add a tuple to this table unless there is a corresponding tuple in the table that contains the primary key. In addition, we cannot delete a tuple from the table that contains the primary key unless the corresponding tuple in the table that contains the foreign key is deleted. The `Department` table has a not null foreign key, i.e., *chair* that references a tuple in the `Professor` table. Referential integrity enforces the inability to insert a tuple to the `Department` table unless the *chair* attribute references an existing tuple in the `Professor` table. Referential integrity of bi-directional relationships is automatically maintained in ODMG 3.0. If an object participating in a relationship needs to be deleted, then any traversal path to that object must also be deleted. For example, when an object of `CourseSection` is deleted, then not only is that object's reference the `Instructor` (i.e., `Professor/TA`) object through the *teacher* traversal path deleted, but also any references in `Instructor` objects to the `CourseSection` object via the *teaches* traversal path must also be deleted. We have deleted an object computed by the OQL query in lambda-DB using %`delete()` function. However, it seems that the object is not removed from relationships and object references, and this may cause dangling pointers. In addition, referential integrity on `ref`s that are in nested tables in the ORDB is not guaranteed because Oracle does not have a mechanism to do so. This integrity could be preserved, e.g., using triggers once the migration process is completed. For XML data, referential integrity carries over with the XML

Schema `key` and `keyref` keywords. The `keyref` keyword specifies a reference to an element specified by `key` or `unique`. In the test, it was found that each *chair* value in the `Department` element must refer to a valid *id* value in a `Professor`.

In summary, in this experiment we ran the selected queries for a subset of the RDB version of the UniDB and for each of the three corresponding target databases to measure the source and target database equivalence. We have loaded the RDB and target databases into their native database systems to check whether or not the results are the same. The queries and their results were applied on a subset of the UniDB to be assessed easily by a user. The user would analyse the sample databases to guarantee that the queries had been designed correctly to give the right results. After we analysed the results of the queries obtained from the four databases, we found that the results were identical. The query results show that the target databases have been generated without redundancy or loss of data. Consequently, this means that the CDM represents a key mediator for converting an existing RDB data into any of the target databases. The CDM has successfully guided the extraction, transformation and loading of data into target databases by facilitating the reallocation of attributes in RDBs to the appropriate values in the target databases. Moreover, many semantics, which have been extracted from the RDB into the CDM, such as relationships and integrity constraints, are tested by the queries and found to be preserved in the target databases. Integrity constraints have been addressed and tested, making our approach superior to its predecessors at this point. However, in object-based systems referential integrity cannot be specified. Key constraints of the RDB are converted into the XML Schema document as `key`/`keyref` and `unique` elements, which can enforce referential integrity and the uniqueness of primary keys in XML instance documents. However, the keys of XML elements, i.e., super-types, which are usually defined in an XML Schema using XPath [Berglund et al., 2007] are not valid for other element(s), i.e., sub-types, which would substitute the super-type. This is because XPath is not schema-aware. In addition, from this experiment, we noticed that a query can work well but may not give the desired results, and this cannot be easily identified if the query is run on a large database.

9.4 Performance Comparison

Target databases may be accessed through the concepts of their data models with a reduced overhead in term of performance compared to an existing RDB. This section describes an experiment designed to explore the efficiency of query processing for an RDB and the equivalent ORDB created in Oracle 11_g DBMS. Although not a direct issue for the research hypothesis, comparing the performance of source and target databases may help the user to decide whether or not they should move into their chosen target database if performance is a deciding factor.

In Experiment II, a set of queries and their results were presented, indicating the intended coverage for each query, regarding the preservation of semantics and data equivalence. The queries were run on a subset of the RDB version of UniDB and the corresponding three targets generated by MIGROX prototype. The results described here are obtained from implementing the first 12 queries of Experiment II on the entire RDB UniDB and its corresponding ORDB generated by the prototype. The elapsed time has been measured for each query. Each RDB query has been executed for unindexed and indexed data. As Oracle supports scoped references, the ORDB has been queried with and without index/scoped references. The results of each query are given, followed by comments on their significance.

Cost metrics: In this experiment, we have measured the query elapsed time as a performance metric. To ensure a secure and stable environment, the experiment has been carried out on a stand-alone isolated PC, so that fluctuations in network activity could not affect query execution. All queries were run with the buffer pool empty as the Oracle system was shut down and restarted for each query. While we were obtaining elapsed times in repeating the query many times, it was found that apart from the first reading, all the subsequent elapsed times were somewhat similar. Thus the average was taken from the second to the fourth time readings. Table 9.2 shows the measured times in seconds for both RDB and ORDB versions of the queries. The table shows the measured time as indexed and unindexed for RDB queries and indexed/scoped and unindexed/unscoped for ORDB queries. The times are shown as variant A/variant B for the last two queries. In addition to measuring elapsed times, the EXPLAIN PLAN statement was used to determine the execution plan that Oracle DBMS follows in performing each query. The EXPLAIN PLAN

statement inserts a row describing each step of the execution plan into a table called PLAN_TABLE. This table contains the necessary metrics, including the cost of executing the query, and CPU and I/O costs for any indexes defined in the table. The PLAN_TABLE gives a good understanding of the operational sequence that Oracle performs to run the queries. Tables 9.3 and 9.4 show performance analysis results of the SET-ELEMENT query execution for the RDB and ORDB, respectively. The output from PLAN_TABLE that Oracle fills as a result of the EXPLAIN PLAN statements for RDB and ORDB queries with indexes and scoped references are provided in Appendixes F and G, respectively.

Query	Relational		Object-relational		rows
	indexed	unindexed	indexed/scoped	unindexed/unscoped	selected
1- SINGLE-EXACT	0.00	0.01	0.00	0.00	1
2- HIER-EXACT	0.00	0.01	0.00	0.01	1
3- SINGLE-METH	0.24	0.25	00.23	496.80	2014
4- HIER-METH	1.03	1.00	0.96	737.61	2788
5- SINGLE-JOIN	1.28	1.21	1.28	1.26	3044
6- HIER-JOIN	0.34	0.39	0.03	0.03	1
7- SET-ELEMENT	0.21	0.14	0.10	0.11	277
8- SET-AND	0.13	0.15	0.12	0.12	277
9- 1HOP-NONE	43.07	43.07	43.13	43.12	75000
10- 1HOP-ONE	0.00	0.03	0.00	0.46	1
11- 1HOP-MANY	0.04	0.07	0.04/0.03	0.07/0.48	318
12- 2HOP-ONE	0.07	0.07	7.22/0.06	61.8/0.31	530
Sum	46.41	46.40	45.94	1279.90	

Table 9.2: Measured times in seconds for queries

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		497	53676	489 (3)	00:00:06
1	NESTED LOOPS		497	53676	489 (3)	00:00:06
2	NESTED LOOPS		497	47215	489 (3)	00:00:06
3	TABLE ACCESS FULL	PERSON	114K	8527K	479 (1)	00:00:06
* 4	INDEX UNIQUE SCAN	KIDS_PK	1	19	0 (0)	00:00:01
* 5	INDEX UNIQUE SCAN	STAFF_PK	1	13	0 (0)	00:00:01
Predicate Information (identified by operation id): 4 - access("P"."ID"="K"."ID" AND "K"."KIDNAME"='boy90') 5 - access("S"."ID"="K"."ID") Note–dynamic sampling used for this statement						

Table 9.3: Plan table for relational SET-ELEMENT query

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		279	28458	240 (1)	00:00:03
* 1	HASH JOIN		279	28458	240 (1)	00:00:03
* 2	TABLE ACCESS FULL	KIDNAMES_STAFF_NT	279	4464	69 (2)	00:00:01
3	TABLE ACCESS FULL	STAFF	20802	1747K	171 (1)	00:00:03
Predicate Information (identified by operation id): 1 - access("K"."NESTED_TABLE_ID"="S"."SYS_NC0001500016$") 2 - filter("K"."KIDNAME"='boy90') Note–dynamic sampling used for this statement						

Table 9.4: Plan table for object-relational SET-ELEMENT query

Database size: We have worked with up to 27.5M of RDB data and corresponding data up to 115M of ORDB in Oracle. The size difference comes from the update statements in the ORDB input files. RDB UniDB table cardinalities (number of tuples) are given in Table 9.1. Although the RDB version of UniDB is a relatively small data set, we have found that it is sufficient to evaluate the DBMS performance using it and its corresponding ORDB data generated by the prototype. RDB data have been loaded to Oracle using SQL*Loader, which is a very efficient data loading tool. It was much faster than loading the script files generated by our prototype. As ORDB object definition files and relationship files contain thousands of "`insert into`" and "`update`" statements, it was expected that loading these files would take much longer than using SQL*Loader, especially for object relationship files. We have loaded the RDB data and ORDB object definition files before creating any indexes since indexes increase the object loading time. Before loading ORDB object relationship files, we created indexes on user-defined object identifers, which speed up the process of establishing relationships among objects.

Indexing: To speed up the response time in query processing, we created other appropriate indexes. Indexes are defined considering the queries and what data would be retrieved. Indexes have been created for elements such as *name, birthdate, zipcode*, etc. Foreign keys are also indexed whereas primary keys have default indexes in Oracle. Nested tables have been indexed on NESTED_TABLE_ID. The `salary()` function, which is used to calculate employee salaries has also been indexed. A list of indexes created for this experiment may be found in Appendix H.

Query 1: SINGLE-EXACT

This query tests exact match look up over a single table. As both relational and object-relational tables have the same number of attributes, tuples and indexes in this query, the result times were identical, as expected. The cost (0.00s) estimates were equal for both queries before and after indexing and scoping (for ORDB queries). The query plans were also equal.

Query 2: HIER-EXACT

This query assesses the system's efficiency in managing queries over inheritance hierarchies. Although indexing/scoping increases the time taken slightly (from 0.00s to 0.01s), all queries performed very similarly with respect to time. As the `union` operation was hidden in the query, the ORDB version was more natural and simple than the RDB query.

Query 3: SINGLE-METH

This query compares performance time for calculating data stored in attributes in the RDB with invoking functions in the ORDB. In the ORDB query, we used the `salary()` function [1] to calculate the salaries of the professors (as shown below in Variant B of the query). Figure 9.9 shows the body of the function, whereas the function definitions of the other sub-types of the `Employee_t` type can be found in Appendix C. Without indexes/scopes, the ORDB query was painfully slow (496.80s). The bad performance, as shown in the plan table, was because of the range scans that have been made by the optimizer to all nested tables in the `Professor` table. These nested tables include `advises_nt`, `teaches_professor_nt` and `kidnames_professor_nt`. To speed up the execution time, these nested tables are indexed on the object identifier, which improves the performance with the time dropping to 11.90s. Even this length of time shows that the ORDB query is still slow, compared to the relational time (0.25s). However, the performance was enhanced considerably further when an index was created on the function. After indexing the function, the ORDB query time (0.23s) shows that the system is more efficient in handling indexed functions, compared to the complex predicates of the RDB query.

Variant B
```
select id, aysalary from professor p where p.salary() >= 145000;
```

```
create or replace type body Professor_t as
 overriding member function salary return number is
 begin
   return (aysalary * (9 + monthsummer) / 9.0 );
 end;
end;
```

Figure 9.9: The `salary()` function for `Professor_t` type

[1] As MIGROX does not map such methods, we have written the body of these methods manually and added them later to their object types.

Query 4: HIER-METH

This query tests the efficiency of the system in invoking indexed functions over inheritance hierarchy. Without indexes/scopes and unindexed function, the ORDB query was very slow (737.61s). Similar to SINGLE-METH, the performance of Oracle improved significantly, with a response time of 0.96s for the ORDB, after the function was indexed, and was then faster than the relational time of 1.03s.

Variant B
```
select s.name, s.street, s.city, s.zipcode from staff s where
s.salary() >= 140000 union select p.name, p.street, p.city, p.zipcode
from professor p where p.salary() >= 140000 union select t.name,
t.street, t.city, t.zipcode from ta t where t.salary() >= 140000;
```

Query 5: SINGLE-JOIN

This query is the baseline test for traditional relational join operations. As the structures of both queries were the same, the query times and the execution plans were similar. Although the system seems slower with indexes (1.28s), the results show that Oracle is efficient in handing join operations in both RDB and ORDB queries.

Query 6: HIER-JOIN

This query tests the efficiency of the system in handling joins among inheritance hierarchies. Executing this query, Oracle was almost 10 times faster with ORDB query compared to the RDB query, with similar performance before and after indexing and scoping with times of 0.03s. The relational times were slower at 0.34s and 0.39s before and after indexing, respectively.

Query 7: SET-ELEMENT

This query tests the system's ability to handle collection data types. The RDB query includes joins among `Person`, `Staff` and `Kids` tables, which make it slower than the ORDB query. The ORDB query performed better than the RDB query, which proves that Oracle is powerful in managing nested tables. An index was created on the object identifier for the `kidnames_staff_nt` nested table and the *kidname* attribute. However, it seems that indexing does not improve the performance and the elapsed time was still similar, although the plan table shows that the nested table is accessed by the index range scan.

Query 8: SET-AND

This query is similar to the SET-ELEMENT with a more complex structure to test the effectiveness of Oracle in handling more complex value-based collections. Although the response times of both queries were close (i.e., 0.13s and 0.15s for the RDB query and 0.12s for the ORDB query) the results show that the system is still efficient in handling value-based collection/sets of data stored in nested tables.

Query 9: 1HOP-NONE

This query tests the system efficiency at processing queries that involve one-hop path expressions. In this query, the entire `Student` table was scanned with the 75000 row selected. The two versions of queries are very close in elapsed time. Although in the ORDB query, path expressions and scoped references were used, Oracle was slightly faster in the RDB query (43.07s) compared to the ORDB query (43.13s). Using scoped references, the system uses the knowledge that the `ref`-based attribute points to an object of a particular type (`Department_t` here). However, indexes and scoped references do not increase performance in the ORDB query.

Query 10: 1HOP-ONE

This query tests how Oracle handles a short path expression. The elapsed times of both RDB and ORDB queries with indexes were similar, whereas with unindexd/unscoped settings, the ORDB query was 15 times slower than the RDB query without an index. As bi-directional relationships are offered in the ORDB schema generated by MIGROX, this query can have another variant (given below), in which the efficiency of the system at handling queries involving a collection of references can be tested (as seen in the subsequent two queries). However, intuitively, as the data required are for a particular student where its related object contains a reference pointing to the department object, it would be better to avoid this variant.

Variant B
```
select s.column_value.id, s.column_value.name, d.deptno,
d.name, d.building from department d, table(d.students) s where
s.column_value.name= 'studentName75001';
```

Query 11: 1HOP-MANY

This query tests the efficiency of Oracle at handling queries that contain collections of references. The ORDB version of this query has two variants. Variant A (given in Experiment II) with `column_value` performed well in either case: indexed/scoped (0.04s) or unindexed/unscoped (0.07s). However, Variant B (given below) with unindexed/unscoped references was somewhat slower than the RDB and the ORDB Variant A queries. The query response time was 0.46s compared to just 0.03s and 0.04s in the other equivalent queries. In other words, it was 16 times slower than the equivalent ORDB query Variant B with an index and scoped references, and 12 times slower than the equivalent RDB query with an index.

Variant B
```
select s.id, s.name from student s where s.major.name = 'deptname1';
```

Query 12: 2HOP-ONE

This query examines Oracle ability in handling queries with longer path expressions. Similar to 1HOP-MANY query, the ORDB version of this query has two variants. The performance of ORDB Variant A (given in Experiment II) was very poor before indexing (61.8s) compared to the RDB and the ORDB Variant B queries given below. Thus, the performance of the Variant A with the selection of two-hop chain set-valued references (an inner collection of references) was very poor. The query starts from a department that contain a set of courses, where each course contains a set of sections. Although the time improved (7.22s) when the references were scoped and indexes created for the identifiers of nested tables, we could not find a way to increase the performance of the Variant A query. However, Variant B using the inverse side of the relationship from sections to courses to the department performed pretty well (0.31s) compared to the ORDB Variant A (61.8s). In addition, Variant B with indexes/scoped references did even better (0.06s) than the RDB version of the query (0.07s).

Variant B
```
select s.semester, s.nostudents, s.course.dept.deptno,
s.course.dept.name from coursesection s where s.roomno = 50;
```

In summary, in this experiment, we ran the first 12 queries used in Experiment II on the RDB UniDB and the corresponding ORDB. We loaded the entire RDB and ORDB into Oracle 11_g to measure the performance for both versions of the queries. All the queries were run with and without indexing, and with and without scoped

references for the ORDB. After comparing and analysing the results, we can draw the following conclusions:

- The relational and object-relational elapsed times are virtually identical for all queries on a single table. Indexing and reference scoping do not improve performance in these kinds of queries.
- In single and hierarchical method queries, the elapsed times are very close. The system performance with ORDB queries improved when the methods were indexed. However, when not indexed, the ORDB query performance was very poor. That is because all nested tables, embedded in accessed object tables, are scanned while invoking the methods.
- The system with the ORDB version of HIER-JOIN query was faster than in the RDB query, verifying that the ORDB outperforms the RDB in handling inheritance and traditional join operations.
- In handling SET-ELEMENT and SET-AND queries, the system was slightly faster with ORDB than with the RDB queries. The results verify that Oracle is more efficient in handling value-based collection data type stored in nested tables. The ORDB with set value-based attributes succeeds over relational joins. Indexing/scoped references make no difference to performance in both versions of the queries.
- By looking at path expression queries, it can be noticed that the elapsed times for RDB and ORDB queries were almost identical. The 1HOP-NONE times were more or less the same in both of the query versions. This is for indexed/unindexed RDB queries and only indexed/scoped ORDB queries. By executing the 1HOP-ONE query, the ORDB scoped short path expression query performed very well like the RDB query. In addition, using `column_value` for dereferencing objects was effective for the 1HOP-MANY ORDB query. However, the time taken for the 2HOP-MANY query with unindexed/unscoped references was obviously slow. Oracle was inefficient in managing queries of two-hop chain of `ref`-based collections. As relationships in the ORDB schema are defined bi-directionally, we used the opposite direction in this query, i.e., the M side of the relationship. For this option with indexing nested tables and scoped references, the query processing performance significantly improved. Therefore, for ORDB

queries with index and reference scoping, Oracle was faster in handling path expressions than in processing the RDB queries.

- The performance of the system is directly affected by the number of tables and attributes in each query, and also by the structure of the query and the number of rows in each table.

- Generally speaking, we can see that in the ORDB Oracle was more efficient in handling queries over inheritance hierarchies, indexed methods, path expressions and set element queries. In addition, the query structure in ORDB queries is more simple and concise than in relational ones. At the end, after having the summation of the reported elapsed times of each set of queries (see last row in Table 9.2), the ORDB efficiency with indexed/scoped data was slightly better than that of the RDB queries. However, the ORDB query with unindexes/unscoped references was painfully slow. The overall time of all RDB queries with indexes was 46.41s and without indexes was 46.40s. The overall time of ORDB queries with indexes/scoped references was 45.94s and with unindexes/unscoped references was 1279.90s. Therefore, the system performance with the RDB queries is not improved when data were indexed, and compared with the corresponding RDB queries, the ORDB queries with indexes/scoped references are slightly more efficient in Oracle 11_g.

9.5 Summary

The effectiveness of MIGROX and its prototype has been evaluated in this chapter. The algorithms of the proposed method were tested in experiments based mainly on database equivalence. Two experiments were setup to validate our approach by examining the differences between a source RDB and each of the three target databases resulting from the use of the prototype. The prototype outputs were given and the overall results of the evaluation discussed.

The correctness of the concepts and the algorithms of MIGROX are checked in Experiment I by comparing the target schemas resulting from the prototype and those generated by existing manual mapping techniques. It was found that the results were very compatible. In addition, using the query-based Experiment II, source and target data equivalences were checked by observing variations in results regarding data

content and integrity constraints.

After applying MIGROX for several RDBs, the results show that the source and target databases are equivalent, and that the MIGROX solution is demonstrably conceptually and practically feasible, efficient and correct. In addition, MIGROX was more comprehensive than existing approaches because it generates a complete database or its schema only, or a single target database or up to three different databases. Besides, integrity constraints have been addressed and tested in our approach, making it superior to its counterparts on this point.

By testing query processing, it was found that system performance depends on various factors, including the size and the type of the databases, the schema and query structure, the number of tuples scanned in each table, indexes and the environment, in which the experiment was carried out.

Chapter 10 closes this dissertation with the conclusions of the research and suggestions for future work.

Chapter 10

Conclusion and Future Work

This chapter presents the conclusion drawn from the research reported in this dissertation. It reviews the research followed by a summary of its major contributions. The chapter then revisits the problems outlined in Chapter 1 and describes open issues and suggestions for further work.

The rest of this chapter is organised as follows. Section 10.1 provides an overview of the dissertation and evaluates the achievement of the research. Its major contributions are summarised in Section 10.2. Section 10.3 draws the main conclusions of the dissertation, and Section 10.4 indicates some areas that may benefit from the research, and suggests directions for future work.

10.1 An Overview of the Dissertation

As mentioned in Section 1.3, the objectives of this research were set as follows:

1. to review existing approaches to the migration of RDBs into object-based and XML databases, considering their capabilities, weaknesses and limitations,
2. to devise a comprehensive solution to the problem of RDB migration, aiming to provide complete migration into richer models including object-based models such as OODB and ORDB, and semi-structured models such as XML,
3. to implement the solution as a prototype, and
4. to evaluate the prototype by critically analysing its results so as to measure the effectiveness of the solution in practice.

After introducing the main concepts of RDBs and the richer types of databases in Chapter 2, the first objective of this research was addressed in Chapter 3. The chapter provided a survey of the existing approaches and techniques for migrating RDBs into richer database models. Based on our analysis of the literature and the open problems mentioned in this chapter and Chapter 1, we proposed a complete method called MIGROX [Maatuk et al., 2008a], which would be able to preserve the semantics of an existing RDB in a CDM for generating an OODB, ORDB and XML.

An overview of the MIGROX solution, the second research objective, was addressed in Chapter 4. The chapter introduced the concepts and assumptions underlying the solution. A detailed description of the MIGROX solution is provided in Chapters 5-7, covering the algorithms for semantic enrichment, schema translation and data conversion.

Addressing the third objective, Chapter 8 described how the prototype of MIGROX was developed by implementing the algorithms and concepts presented in Chapters 4-7. The prototype performs the process of database migration and generates three different output databases from an RDB.

Chapter 9 discussed how an experimental study was conducted to evaluate the prototype, thereby addressing the fourth objective of the research. The correctness of the CDM and the database migration algorithms was checked by comparing the equivalence between the source RDBs and the target databases generated by the prototype. In addition, the outputs of the prototype were compared with the results produced with existing approaches. The evaluation was performed by observing variations in the results regarding schema, data and integrity constraints.

10.2 A Summary of the Contributions

The main aim of this research was to devise a solution for migrating an RDB into object-based and semi-structured models. This aim has led to the achievement of three major contributions: the review of relevant literature, the solution for RDB migration and research publications. These contributions are summarised as follows:

The Review of Relevant Literature: The literature shows that the existing approaches concerned with database conversion are: 1) viewing objects on top of

RDBs where data is processed in object/XML form and stored in relational form, 2) database integration where a gateway is used on top of multiple heterogeneous databases to support a single view, and 3) database migration where an RDB is migrated completely into its equivalents. In this dissertation, we devised a solution for migrating RDBs into three equivalent target databases according to recent standards since no existing research has achieved such RDB migration, covering both schema translation and data conversion.

The investigation of the relevant literature has shown that viewing objects on top of existing RDBs and establishing gateways to access existing data only for data retrieval purposes solves the problems neither of mismatches between different paradigms nor of the preservation of RDB data semantics. Besides, it seems that existing work does not provide a complete solution for more than one target database for either schema or data conversion. In addition, none of the existing work can be considered as a method for migrating an RDB into an ORDB. Some semantics such as inheritance, are not considered in some work mainly due to a lack of support for such semantics either in source or target data models. Less effort has been devoted to using standards as target models, e.g., ODMG 3.0 and SQL4. Such standards provide better levels of semantic preservation and portability. Although known conceptual models like ER and UML, in addition to several specific models such as the BLOOM model [Abelló et al., 1999], may be used as an intermediate stage for enrichment during RDB migration, however, we argue that either they are not appropriate for the characteristics of more than one target data model, or do not support data representation. In addition, the vast majority of work has migrated RDBs directly into target databases, where the latter either look like flat RDBs or have deep levels of clustering/nesting. In this kind of work, object-based model features and the hierarchical form of the XML model are usually missed, or well-designed target databases could not be guaranteed in term of data redundancy.

A Solution for RDB Migration: A significant contribution of this dissertation is to offer a precise description of a solution for migrating an RDB into object-based and semi-structured models in the form of MIGROX, which has the following features.

- It offers a CDM as an intermediate stage for better preservation of integrity constraints and data semantics. Using the CDM, a new target database can be generated without referring to an existing RDB each time another target

database needs to be generated. This provides the reading and enrichment of an RDB only once for multiple subsequent usages.

- It translates the CDM into an object-based and XML schema according to internationally recognised standards such as ODMG 3.0, SQL4 and XML Schema, and data definition languages, leading to more portability and flexibility. Algorithms have been developed, each of which consists of a set of rules, which describe how to translate each construct of the CDM into a specific construct in the target schema.

- It provides data conversion techniques, which automatically convert an existing RDB data into any of the target databases based on the CDM. In other words, the CDM determines and manages the conversion of data into the target formats.

- It is the first approach designed for migrating RDBs into ORDBs [Maatuk et al., 2010].

- It has been implemented as a prototype. The implementation helps to demonstrate the effectiveness of the proposed solution and its concepts. The prototype facilitates the migration process and, through its outputs, illustrates that MIGROX and its three phases can be practically executed. The current prototype can be considered as a basis for an integrated database migration tool.

- An experimental study for evaluating the prototype of MIGROX has been conducted. The experiments have been designed to validate the solution, demonstrating the correctness of the CDM and the algorithms, and the completeness of the schema translation and data conversion rules. The results of the experimental study demonstrate that the MIGROX solution is feasible, efficient and correct both conceptually and practically. The results obtained can be summarised as follows.

  - The target schemas produced by MIGROX and existing manual algorithms were comparable, showing that MIGROX and existing manual algorithms compared with it preserve equivalence in translations.

- Three target databases equivalent to the input RDB have been generated by MIGROX without redundancy or loss of data. Analysing the information resulting from querying the four databases (i.e., the RDB, OODB, ORDB and XML), it has been found that the results were identical. In addition, integrity constraints have been tested, making MIGROX superior to its predecessors in semantics preservation.

- ORDB and RDB queries run on the same DBMS, i.e., Oracle 11_g were compared. The ORDB queries are generally more efficient, and the query structure is more simple and concise than that of the relational ones.

Publications: Some parts of the research reported in this dissertation have been published as follows.

- Abdelsalam Maatuk, M. Akhtar Ali and B. Nick Rossiter. Relational Database Migration: A Perspective. In Bhowmick, S. S., Küng, J. and Wagner, R., editors, *DEXA*, volume 5181 of Lecture Notes in Computer Science, pages 676–683, Springer (2008).

- Abdelsalam Maatuk, M. Akhtar Ali and B. Nick Rossiter. An Integrated Approach to Relational Database Migration. In *the Proceeding of International Conference on Information and Communication Technologies-2008* (IC-ICT '08), pages 01-06, Bannu, Pakistan (2008).

- Abdelsalam Maatuk, M. Akhtar Ali and B. Nick Rossiter. Semantic Enrichment: The First Phase of Relational Database Migration. In *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering* (CIS^{2}E 08), pages 6pp, Bridgeport, USA (2008).

- Abdelsalam Maatuk, M. Akhtar Ali and B. Nick Rossiter. Re-engineering Relational Databases: The Way Forward. In *Journal of Computing and Information Technology* (CIT), pages 14pp (2009, submitted).

- Abdelsalam Maatuk, M. Akhtar Ali and B. Nick Rossiter. Converting Relational Databases into Object-relational Databases. In *Journal of Object Technology* (JOT), pages 17pp (2010, in press).

10.3 Conclusions

The main conclusions that can be drawn from this research are as follows.

- This dissertation addresses the question of whether an existing RDB can be automatically migrated into object-based and XML databases. Several challenges could arise when the database migration process is aimed at multiple target databases, which are fundamentally different and have different design characteristics. The dissertation described a solution to this question which is superior to existing proposals as it produces three different output databases. The solution has been implemented and evaluated.

- MIGROX is the first approach that achieves a solution to the problem of migrating RDBs into ORDBs. In addition, it is the first approach to migrate RDBs into more than one comparatively newer and richer target platform.

- MIGROX is one of the few methods which cover schema translation and data conversion with data semantics preserved, including integrity constraints, and association, aggregation and inheritance relationships.

- MIGROX is supported by an approach to semantic enrichment in which the necessary data semantics of a given RDB are inferred and enhanced to generate a CDM, which provides a description of the existing RDB's implicit and explicit semantics. The CDM is a sound source of semantics and is a well organised data model, forming the starting point for the remaining phases of database migration. Its specifications are based on the similarities among object-based and XML data models. These similarities produce natural correspondences that can be exploited to bridge the semantic gap among diverse data models. This provides a basis for the CDM, being used as an intermediate representation when migrating an RDB into more than one target database. In addition to considering the most important characteristics of the target models, the CDM preserves all data semantics that can be extracted from an RDB. Moreover, the CDM acts as a key mediator for converting an existing RDB data into target databases.

- The algorithms of MIGROX have been implemented as a prototype, which successfully generated the target databases. Java is used for basic coding. The

ability of Java to connect to RDBs and the reusability of code makes it preferable for use through JDBC. The JDBC metadata classes and their methods were useful and saved time in providing full access to metadata and deriving information describing the content of the RDBs.

- MIGROX is one of only a few database migration methods that have been evaluated in terms of the equivalence of input and output databases. We used an experimental study to validate the solution by testing the hypotheses developed and introduced in Section 9.2. This has been performed by comparing the RDBs input into the prototype with its output, and comparing the output databases that the prototype generates with the target databases generated by existing techniques. Since we work with real databases and the target databases lack a mathematical foundation compared to the source RDBs, we believe that the experimental study is an appropriate approach to validate the proposed solution and test the hypotheses. The experiments explored the following questions:

  1. Have the target schemas generated by the prototype preserved the data semantics of the source RDB?

  2. Are the data and integrity constraints of the source and target databases equivalent?

  When MIGROX was evaluated by comparing the output schemas of its prototype with the ones produced by existing work, it was found that both sets of schemas were comparable. The CDM preserves and enhances the metadata of existing RDBs, and is translatable into any of the three target database schemas. Therefore, the algorithms for translating CDM into the target schema are empirically shown to be correct. In addition, MIGROX was more comprehensive than its counterparts since it generates up to three different databases.

  The results of queries obtained from source and target databases were analysed and it was found that both set of results were identical. Therefore, we conclude that both the source and the target databases, generated by MIGROX, are equivalent. This means that the CDM has successfully guided the extraction, transformation and loading of RDB data into target databases, facilitated the reallocation of attribute values in RDBs to the appropriate domain in the target databases. Moreover, many of the semantics, e.g., relationships and integrity

constraints, extracted from the RDB were found to be preserved in the target databases.

Although it is not conceptually related to the research hypotheses, the efficiency of database systems in terms of RDB and ORDB query processing is assessed. Comparing RDB queries with their equivalents in an ORDB implemented on Oracle 11_g, it was found that the system is more efficient in handling ORDB queries over the inheritance hierarchy, indexed methods, path expressions and set element queries. In addition, the structure of ORDB queries is more simple and concise than the relational ones. The performance of the system is directly affected by the number of tables and attributes in each query, as well as by the structure of the query and the number of rows in each table.

10.4 Applications and Further Work

10.4.1 Applications

MIGROX and the concepts it contains might be useful for many applications such as the following:

- In database design, conceptual data models provide different concepts and diagrams to facilitate the development process from requirements specification. Based on the analysis of data requirements specification, entity types, relationship types, attributes and keys are identify and represented using a conceptual data model, from which a logical data model is derived. The logical model is then translated into a physical data model for implementation. The techniques, developed here for the schema translation phase of MIGROX, for translating data semantics represented in the CDM into target physical schemas, are applicable for this purpose. However, the CDM may need to be extended to cover dynamic aspects, i.e., operations supported by object-based models. In addition, the CDM (when extended) could be used to represent the conceptual data-model used in CASE tools (e.g., Oracle Designer and Rational Rose) and the schema translation rules could be used to generate physical schemas for object-based/XML and relational implementation.

- The sematic enrichment approach of MIGROX is not only essential for database

migration, but also useful for database redesign and maintenance, and possibly for database integration. During database redesign and maintenance, some semantics might not be captured or difficulty identified from schemas or DBMSs, for example, the determination of occurrence among tuples in relations participating in relationships. The technique for identifying relationships and determining the cardinalities among tuples from database instances, which has been developed for the semantic enrichment phase of MIGROX, is useful to make full use of this. Database integration would be best performed at a canonical model level. The CDM model is sufficiently enriched to represent the semantics expressed in the local schemas for integration.

- The essential concepts for teaching database courses to university students are database design models (e.g., ER), traditional logical and physical models (e.g., relational models) and schema transformation from conceptual models into specific implementation models. However, it might be important for those who already have an essential understanding of traditional modeling to acquire knowledge about techniques for translating RDBs into object-based and XML databases. MIGROX could be used, e.g., in more advanced database courses, as a comparative approach to RDBs with diverse types of more recent database models in terms of database design and migration.

- Several benchmarks have been designed to test different performance aspects of database systems [Carey et al., 1993, 1997; Kurt and Atay, 2002]. Benchmarks provides metrics on how to test the functionality and performance of systems by querying the data stored in them. Databases generated by MIGROX can be used in benchmarks to evaluate the performance of query operations over contemporary DBMSs.

10.4.2 Further Work

The process of information system re-engineering is complex and involves a wide range of tasks associated with the understanding and transformation of existing systems. Much more work in these areas is required. This dissertation contributes to the field of database migration, including schema translation and data conversion. Various relevant aspects might be studied in order to fully benefit from the solution reported in this dissertation. In this section, some suggestions for future work are outlined, which

are either complementary to MIGROX, adding more functionality, or extensions of this research.

MIGROX is restricted by a number of factors such as assumptions on which is based, including incompatibilities among standards and database products and the level of automation that the method assumes.

- MIGROX mainly focuses on migrating RDBs with the corresponding data dictionary that contains all metadata information, and hence the existence of modern RDBMSs is assumed. However, not all existing RDBs can provide such information stored in data dictionaries, and thus the application of existing database reverse engineering techniques is required. A connection between MIGROX and existing database reverse engineering techniques could be an issue for further work to facilitate the gathering of sematic information when data dictionaries are missing. Besides, the migration process should consider not only relations but also views of the input RDBs.

- It should be considered that as yet there are no implementations of all standard constructs. Collection types in SQL4 such as `row` and `set`, and also inheritance among typed tables are not supported in all DBMSs. In our prototype, the collection types in ORDB are implemented using nested tables in Oracle. In addition, since Oracle does not support data or table inheritance, two different but semantically equivalent ORDB schemas can be generated, based on the root or leaves representations. Although the MIGROX prototype can produce both types of inheritance representations, the latter was used in our experiments. Another issue is that Oracle (and also Lambda-DB) cannot maintain referential integrity for collections of `OID`s. Oracle has no mechanism to maintain referential integrity on `ref`s that are in nested tables in the ORDB. This integrity could be preserved, e.g., by using the triggers, once the migration process is completed. In addition, the keys defined for XML elements may not be valid for other elements, which would substitute them in the instance document. This is because keys in XML Schema are defined using XPath 2.0, which is not schema-aware.

- MIGROX assumes that the process of database migration is to be conducted automatically. The user interacts with the system only to give more meaning

to the relationship names, generated automatically by the prototype. The flexibility of the database migration process to allow help from the user, and in resuming the loading of the target databases generated after interruption, is worth further investigation. Whatever the size of the RDB being migrated, the current prototype offers the generation of the corresponding target schemas or the complete database in one go, and stores the results in a secondary storage. In addition, the corresponding CDM is stored only as an in-memory representation for further reference during schema translation and data conversion. The prototype can be supported by undo/redo operations and the CDM can be stored for use on another occasion or to be edited. In addition, the question of how to proceed with the process when the system crashes or other interruptions occur needs to be discussed. Algorithms for loading bulk data into an OODB, taking into account process resumptions/system crashes have been proposed [Wiener and Naughton, 1994; Wiener, 1996].

MIGROX can be further extended in several possible ways:

- Although the current prototype implements all the concepts and algorithms of MIGROX, it does not provide a graphical user interface. We argue that it is necessary to improve the current prototype with a visual user-friendly interface in the form of a complete database migration tool that can connect other RDBMSs rather than just Oracle. This might increase the acceptance of object-based and XML databases in the community, particularly among those who already have knowledge of RDBs.

- Migration of database applications involves the conversion of schema, data and application programs into a target platform. The SQL query and update operations embedded in application programs need to be translated into their corresponding versions in the target environment. One such technique for mapping RDB update operations into their OODB equivalent has been proposed [Zhang and Fong, 2000]. Krishnamurthy et al. [2004] consider the problem of query translation between SQL and XQuery. Studying how to translate RDB SQL queries into their equivalents in the target databases, based on the framework described in this dissertation, is an interesting open research problem.

- A complete system should support other standards as target models, such as

DTD [DTD, 2009] and Java Data Objects (JDO) [JDO, 2009]. The DTD is another XML standard from W3C used for many applications. The JDO is a Java application program interface for handling persistent objects. It is a standard as part of the Sun Java Community, which can work with RDBs, object-based databases, XML and others. The JDO technology is used to directly store Java domain model instances into the database.

- Although most concepts of MIGROX are defined using semi-formal notation, we use an experimental study to demonstrate its correctness. The experimental study has become an acceptable approach to testing hypotheses and validating solutions. However, it might be possible to improve this approach and validate MIGROX mathematically. As a consequence, the rules for schema translation and data conversion need to be revised using more formal expressions.

- Another interesting issue might be to setup a benchmark to explore system efficiency on query processing between the RDB version of UniDB of the BUCKY benchmark [Carey et al., 1997] and the equivalent three target databases generated by MIGROX.

RDB migration is an important topic when migrating into newer and richer database environments. This research work represents an important advancement in the area of database migration, showing that development of a canonical model facilitates the migration of schema and data of RDBs into richer databases models.

Appendix A

Attribute Data Type Mapping

CDM/SQL4	ODMG 3.0	XML Schema
char, varchar, varchar2, longvarchar (length >1)	string	string
char, varchar, varchar2, longvarchar (length = 1)	char	string
integer	float	decimal
binary, varbinary, longvarbinary	byte	byte
bit	boolean	boolean
number (data precision = null)	long	decimal
number (data precision ≠ null)	float	decimal
decimal, numeric	decimal	decimal
date	date	date
time	time	time
timestamp	timestamp	timestamp
interval	interval	interval
BLOB	BLOB	BLOB
CLOB	CLOB	CLOB

Table A.1: Data type mapping from CDM to target models

Appendix B

Specification of RDB UniDB

This Appendix contains the sample RDB UniDB, including the description of schema, keys, CTL files, and data instances.

B.1 Description of the schema

```
create table department (
deptno number(3), name varchar(12) not null, building char(8), budget integer, chair
integer not null, latitude number(3), longitude number(3));

create table course (
deptno number(3), courseno number(3) not null, name varchar(16), credits number(1));

create table person (
id integer, name varchar(20) not null, street char(15), city char(9), state varchar(20),
zipcode char(9), birthdate date, picture char(7), latitude number(5),
longitude number(5));

create table student (
id integer, studentno integer, majordept number(3) not null, advisor integer not null);

create table employee (
id integer, dept number(3) not null, datehired date, status number(2));

create table instructor(id integer);

create table staff ( id integer, annualsalary integer);

create table professor (
id integer, aysalary integer, monthsummer number(1));

create table ta (
id integer, semestersalary integer, apptfraction number(3,2));

create table kids ( id integer, kidname varchar(8) );

create table coursesection (
deptno number(3) , courseno number(3) , sectionno number(1), instructorid integer,
semester number(1), textbook char(12), nostudents integer,
building char(8), roomno number(2));
```

```
create table enrolled (
studentid integer, deptno number(3), courseno number(3), sectionno number(1),
semester number (1), grade varchar(3));
```

B.2 Key definitions

```
rem primary keys

alter table department add constraint dept_pk primary key (deptno);
alter table person add constraint person_pk primary key (id);
alter table employee add constraint employee_pk primary key (id);
alter table student add constraint stud_pk primary key (id);
alter table instructor add constraint instructor_pk primary key (id);
alter table staff add constraint staff_pk primary key (id);
alter table professor add constraint prof_pk primary key (id);
alter table ta add constraint ta_pk primary key (id);
alter table course add constraint course_pk primary key (deptno, courseno);
alter table coursesection add constraint coursesection_pk primary key
(deptno, courseno, sectionno, semester);
alter table kids add constraint kids_pk primary key (id, kidname);
alter table enrolled add constraint enrolled_pk primary key
(studentid, deptno, courseno, sectionno, semester);

rem foreign keys

alter table department add constraint department_fk foreign key (chair) references
professor;
alter table employee add constraint employee_fk foreign key (dept) references
department;
alter table employee add constraint employee_fk2 foreign key (id) references
person;
alter table student add constraint student_fk2 foreign key (id) references
person;
alter table student add constraint student_fk foreign key (majordept) references
department;
alter table student add constraint std_adv_fk foreign key (advisor) references
professor;
alter table staff add constraint staff_fk2 foreign key (id) references
employee;
alter table instructor add constraint instructor_fk2 foreign key (id) references
employee;
alter table professor add constraint professor_fk3 foreign key (id) references
instructor;
alter table ta add constraint ta_fk4 foreign key (id) references
instructor;
alter table course add constraint course_fk foreign key (deptno) references
department;
alter table coursesection add constraint coursesection_fk foreign key
(deptno, courseno) references course;
```

```
alter table coursesection add constraint coursesection_fk1 foreign key
(instructorid) references instructor;
alter table kids add constraint kids_fk2 foreign key (id) references employee;
alter table enrolled add constraint enrolled_fk1 foreign key (studentid) references
student;
alter table enrolled add constraint enrolled_fk2 foreign key
(deptno, courseno, sectionno, semester) references coursesection;
```

B.3 Description of CTL files

LOAD DATA
INFILE ‘courses.txt’
INTO TABLE COURSE
FIELDS TERMINATED BY ”,”
(
DEPTNO , COURSENO, NAME ENCLOSED BY ‘”’, CREDITS
)
LOAD DATA
INFILE ‘SECTIONS.txt’
INTO TABLE COURSESECTION
FIELDS TERMINATED BY ”,”
(
DEPTNO, COURSENO, SECTIONNO, INSTRUCTORID, SEMESTER, TEXTBOOK ENCLOSED BY ‘”’, NOSTUDENTS, BUILDING ENCLOSED BY ‘”’, ROOMNO
)
LOAD DATA
INFILE ‘depts.txt’
INTO TABLE DEPARTMENT
FIELDS TERMINATED BY ”,”
(
DEPTNO, NAME ENCLOSED BY ‘”’, BUILDING ENCLOSED BY ‘”’, BUDGET, CHAIR, LATITUDE, LONGITUDE
)
LOAD DATA
INFILE ‘persons.txt’
INTO TABLE PERSON
FIELDS TERMINATED BY ”,”
(
ID, NAME ENCLOSED BY ‘”’, STREET ENCLOSED BY ””’, CITY ENCLOSED BY ‘”’, STATE ENCLOSED BY ‘”’, ZIPCODE ENCLOSED BY ‘”’, BIRTHDATE date ”mm/dd/yyyy”, PICTURE ENCLOSED BY ‘”’, LATITUDE, LONGITUDE
)
LOAD DATA
INFILE ‘students.txt’
INTO TABLE STUDENT
FIELDS TERMINATED BY ”,”
(
ID, STUDENTNO, MAJORDEPT, ADVISOR
)

```
LOAD DATA
INFILE ‘employees.txt’
INTO TABLE EMPLOYEE
FIELDS TERMINATED BY ”,”
(
ID, DEPT, DATEHIRED date ”mm/dd/yyyy”, STATUS
)
LOAD DATA
INFILE ‘staffs.txt’
INTO TABLE STAFF
FIELDS TERMINATED BY ”,”
(
ID, ANNUALSALARY
)
LOAD DATA
INFILE ‘instructors.txt’
INTO TABLE INSTRUCTOR
FIELDS TERMINATED BY ”,”
(
ID
)
LOAD DATA
INFILE ‘profs.txt’
INTO TABLE PROFESSOR
FIELDS TERMINATED BY ”,”
(
ID, AYSALARY, MONTHSUMMER
)
LOAD DATA
INFILE ‘tas.txt’
INTO TABLE TA
FIELDS TERMINATED BY ”,”
(
ID, SEMESTERSALARY, APPTFRACTION
)
LOAD DATA
INFILE ‘enrolled.txt’
INTO TABLE ENROLLED
FIELDS TERMINATED BY ”,”
(
STUDENTID, DEPTNO, COURSENO, SECTIONNO, SEMESTER, GRADE ENCLOSED BY ‘”’
)
LOAD DATA
INFILE ‘kids.txt’
INTO TABLE KIDS
FIELDS TERMINATED BY ”,”
(
ID, KIDNAME ENCLOSED BY ‘”’
)
```

B.4 Sample of data instances

Person

1,"staffName1", "xxxx_streetname", "city_name", "Oklahoma", "41421",11/13/1989, "picture", 362, 27
2,"staffName2", "xxxx_streetname","city_name", "Oregon", "56429",12/9/1957, "picture", 1782, 1530
3,"staffName3", "xxxx_streetname","city_name", "Florida", "18456",6/28/1983,"picture", 1011, 42
59247,"professorName59247", "xxxx_streetname", "city_name", "Utah","98075", 3/8/1971, "picture", 1211, 98
65805,"professorName65805", "xxxx_streetname", "city_name", "Minnesota", "92344", 10/30/1983, "picture", 1337, 60
65990,"studentName65990", "xxxx_streetname", "city_name", "Vermont", "24609", 4/16/1941, "picture", 579, 534
75001,"studentName75001", "xxxx_streetname", "city_name", "Georgia", "47880",7/21/1985, "picture", 1464, 179
75051,"studentName75051", "xxxx_streetname", "city_name", "Florida", "63335",12/18/1956, "picture", 71, 755
117525,"studentName117525", "xxxx_streetname", "city_name", "Oklahoma", "97695", 8/27/1952, "picture", 1619, 900
108981,"studentName108981", "xxxx_streetname", "city_name", "West_Virginia", "83589", 9/2/1965, "picture", 1867, 985
47397,"studentName47397", "xxxx_streetname", "city_name", "Iowa","64177", 10/12/1970, "picture", 1043, 486
71661,"studentName71661", "xxxx_streetname", "city_name", "Idaho","64177", 4/17/1952, "picture", 377, 859
47254,"professorName47254", "xxxx_streetname", "city_name", "Pennsylvania", "34306", 12/3/1981, "picture", 1904, 1565
54453,"professorName54453", "xxxx_streetname", "city_name", "Virginia", "23403", 4/4/1971, "picture", 1111, 1753

Course

1, 124, "coursename124", 1
1, 126, "coursename126", 4
1, 112, "coursename112", 1
220, 122, "coursename122", 2
220, 136, "coursename136", 3
220, 101, "coursename101", 4
10, 110, "coursename110", 3
10, 105, "coursename105", 4
25, 121, "coursename121", 4
25, 135, "coursename135", 3

Department

1, "deptname1", "building", 6000000, 54453, 47, 6
10, "deptname10", "building", 5000000, 59247, 74, 38
220, "deptname220", "building", 6000000, 65805, 23, 37
25, "deptname25", "building", 8000000, 47254, 55, 12

CourseSection

1, 126, 2, 54453, 2, "textbookname", 20, "building", 39
220, 122, 2, 65990, 2, "textbookname", 20, "building", 70
220, 136, 2, 54453, 2, "textbookname", 20, "building", 91
220, 101, 2, 65805, 1, "textbookname", 20, "building", 50
220, 101, 2, 65990, 2, "textbookname", 20, "building", 50
10, 105, 1, 65990, 2, "textbookname", 20, "building", 75
1, 112, 2, 59247, 1, "textbookname", 20, "building", 10
25, 135, 1, 47397, 2, "textbookname", 20, "building", 13
25, 121, 2, 54453, 2, "textbookname", 20, "building", 43

Enrolled
71661, 1, 126, 2, 2, "AB"
71661, 220, 122, 2, 2, "F"
47397, 220, 136, 2, 2, "D"
65990, 220, 101, 2, 2, "AB"
75001, 10, 105, 1, 2, "A"
75001, 1, 112, 2, 1, "AB"
75051, 25, 121, 2, 2, "C"
75051, 25, 135, 1, 2, "B"
117525, 220, 101, 2, 2, "A"
108981, 220, 101, 2, 1, "AB"

Employee
1, 1, 11/28/1955, 3
2, 1,11/1/1966, 2
3, 10, 8/30/1942, 2
59247, 220, 8/8/1958, 9
65805, 220, 4/19/1943, 9
47254, 25, 1/9/1982, 0
54453, 10, 5/14/1970, 9
65990, 220, 9/29/1987, 6
47397, 1, 11/27/1963, 2
71661, 25, 11/27/1963, 9

Instructors
65805
65990
59247
47397
71661
47254
54453

Kids
1, "boy90"
1, "girl90"
2, "boy62"
2, "girl62"
3, "boy90"
3, "girl29"
59247, "boy90"
59247, "girl39"
65805, "boy99"
65805, "girl99"
65805, "girl877"
65805, "girl1057"

Student
65990, 454356786, 1, 54453
75001, 503582122, 1, 59247
75051, 112339778, 10 , 59247
117525, 374361804, 10, 47254
108981, 75613281, 220, 65805
47397, 651921317, 25, 47254
71661, 974503561, 25, 65805

Professor
59247, 115000, 2
65805, 127000, 3
47254, 127000, 1
54453, 111000, 3

Staff
1, 83000
2, 33000
3, 47000

TA
65990, 12000, 0.55
47397, 16000, 0.45
71661, 17000, 0.65

Appendix C

Databases Generated by MIGROX

This Appendix contains the OODB, ORDB and XML generated by MIGROX from the subset of the RDB UniDB given in Appendix B.

C.1 ODMG 3.0 OODB of UniDB

C.1.1 ODMG 3.0 ODL schema

```
module OODB {
class Course (extent Courses) {
   attribute long courseno;
   attribute string name;
   attribute long credits;
   attribute set<Coursesection> sections;
};
class Person (extent Persons key id){
   attribute long id;
   attribute string name;
   attribute string street;
   attribute string city;
   attribute string state;
   attribute string zipcode;
   attribute string birthdate;
   attribute string picture;
   attribute long latitude;
   attribute long longitude;
};
class Coursesection (extent Coursesections){
   attribute long sectionno;
   attribute long semester;
   attribute string textbook;
   attribute long nostudents;
   attribute string building;
```

```
   attribute long roomno;
   relationship set<Enrolled> students inverse Enrolled::section;
   relationship Instructor teacher inverse Instructor::teaches;
};
class Department (extent Departments key deptno){
   attribute long deptno;
   attribute string name;
   attribute string building;
   attribute long budget;
   attribute long latitude;
   attribute long longitude;
   attribute set<Course> offers;
   relationship set<Employee> employees inverse Employee::worksin;
   relationship set<Student> students inverse Student::major;
   relationship Professor chair inverse Professor::leads;
};
class Employee extends Person (extent Employees) {
   attribute string datehired;
   attribute long status;
   attribute set<string> kidnames;
   relationship Department worksin inverse Department::employees;
};
class Enrolled (extent Enrolleds) {
   attribute string grade;
   relationship Coursesection section inverse Coursesection::students;
   relationship Student student inverse Student::taken;
};
class Instructor extends Employee (extent Instructors) {
   relationship set<Coursesection> teaches inverse Coursesection::teacher;
};
class Professor extends Instructor (extent Professors) {
   attribute long aysalary;
   attribute long monthsummer;
   relationship Department leads inverse Department::chair;
   relationship set<Student> advises inverse Student::advisor;
};
class Staff extends Employee (extent Staffs) {
   attribute long annualsalary;
};
class Student extends Person (extent Students) {
   attribute long studentno;
   relationship set<Enrolled> taken inverse Enrolled::student;
   relationship Department major inverse Department::students;
   relationship Professor advisor inverse Professor::advises;
};
class Ta extends Instructor (extent Tas) {
   attribute long semestersalary;
   attribute float apptfraction;
};
};
```

C.1.2 The OODB Makefile

```
include /home/makhtarali/lambda-DB/ldb.include
all: populate

# build the user database and catalog
build:
$(ODLB) -build

# compile the ODL schema ooschema.odl
ooschema.o: ooschema.odl
$(ODL) ooschema.odl
$(GCC) $(ODMGFLAGS) -c ooschema.cc -o ooschema.o

# compile the OQL file populate.oql that populates the database
populate: populate.oql ooschema.o
$(OQL) populate.oql
$(GCC) $(ODMGFLAGS) populate.tmp.cc ooschema.o (LIBS) -o populate
# ./populate # populate the database with data
clean:
/bin/rm -f *.o *  ooschema.sdl ooschema.cc ooschema.h *.tmp.* populate query core
```

C.1.3 The OODB OIF data file

```
/*****************************************************************************
*
* Populating the BUCKY database
* Programmer: A. Maatuk
* Date: 06/11/2008
*
*****************************************************************************/
#include <odmg_main.h>
%module OODB;
int main (int argc,char* argv[])
{ %initialize;
%begin;

// Object definitions
// Course file
%c1124 := persistent Course (courseno: 124, name: "coursename124", credits: 1);
%c1126 := persistent Course (courseno: 126, name: "coursename126", credits: 4);
%c1112 := persistent Course (courseno: 112, name: "coursename112", credits: 1);
%c220122 := persistent Course (courseno: 122, name: "coursename122", credits: 2);
%c220136 := persistent Course (courseno: 136, name: "coursename136", credits: 3);
%c220101 := persistent Course (courseno: 101, name: "coursename101", credits: 4);
%c10110 := persistent Course (courseno: 110, name: "coursename110", credits: 3);
%c10105 := persistent Course (courseno: 105, name: "coursename105", credits: 4);
%c25121 := persistent Course (courseno: 121, name: "coursename121", credits: 4);
%c25135 := persistent Course (courseno: 135, name: "coursename135", credits: 3);
```

// **Coursesection file**

%cc112622 := persistent Coursesection (sectionno: 2, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 39);
%cc22012222 := persistent Coursesection (sectionno: 2, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 70);
%cc22013622 := persistent Coursesection (sectionno: 2, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 91);
%cc22010121 := persistent Coursesection (sectionno: 2, semester: 1, textbook: "textbookname", nostudents: 20, building: "building", roomno: 50);
%cc22010122 := persistent Coursesection (sectionno: 2, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 50);
%cc1010512 := persistent Coursesection (sectionno: 1, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 75);
%cc111221 := persistent Coursesection (sectionno: 2, semester: 1, textbook: "textbookname", nostudents: 20, building: "building", roomno: 10);
%cc2513512 := persistent Coursesection (sectionno: 1, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 13);
%cc2512122 := persistent Coursesection (sectionno: 2, semester: 2, textbook: "textbookname", nostudents: 20, building: "building", roomno: 43);

// **Department file**

%d1 := persistent Department (deptno: 1, name: "deptname1", building: "building", budget: 6000000, latitude: 47, longitude: 6);
%d10 := persistent Department (deptno: 10, name: "deptname10", building: "building", budget: 5000000, latitude: 74, longitude: 38);
%d220 := persistent Department (deptno: 220, name: "deptname220", building: "building", budget: 6000000, latitude: 23, longitude: 37);
%d25 := persistent Department (deptno: 25, name: "deptname25", building: "building", budget: 8000000, latitude: 55, longitude: 12);

// **Enrolled file**

%e71661112622 := persistent Enrolled (grade: "AB");
%e7166122012222 := persistent Enrolled (grade: "F");
%e4739722013622 := persistent Enrolled (grade: "D");
%e6599022010122 := persistent Enrolled (grade: "AB");
%e750011010512 := persistent Enrolled (grade: "A");
%e75001111221 := persistent Enrolled (grade: "AB");
%e750512512122 := persistent Enrolled (grade: "C");
%e750512513512 := persistent Enrolled (grade: "B");
%e11752522010122 := persistent Enrolled (grade: "A");
%e10898122010121 := persistent Enrolled (grade: "AB");

// **Staff file**

%s1 := persistent Staff (id: 1, name: "staffName1", street: "xxxx_streetname", city: "city_name", state: "Oklahoma", zipcode: "41421", picture: "picture", latitude: 362, longitude: 27, status: 3, kidnames: set("boy90", "girl90"), annualsalary: 83000);
%s2 := persistent Staff (id: 2, name: "staffName2", street: "xxxx_streetname", city: "city_name",

state: "Oregon", zipcode: "56429", picture: "picture", latitude: 1782, longitude: 1530, status: 2, kidnames: set("boy62", "girl62"), annualsalary: 33000);
%s3 := persistent Staff (id: 3, name: "staffName3", street: "xxxx_streetname", city: "city_name", state: "Florida", zipcode: "18456", picture: "picture", latitude: 1011, longitude: 42, status: 2, kidnames: set("boy90", "girl29"), annualsalary: 47000);

// **Professor file**

%p59247 := persistent Professor (id: 59247, name: "professorName59247", street: "xxxx_streetname", city: "city_name", state: "Utah", zipcode: "98075", picture: "picture", latitude: 1211, longitude: 98, status: 9, kidnames: set("boy90", "girl39"), aysalary: 115000, monthsummer: 2);
%p65805 := persistent Professor (id: 65805, name: "professorName65805", street: "xxxx_streetname", city: "city_name", state: "Minnesota", zipcode: "92344 ", picture: "picture", latitude: 1337, longitude: 60, status: 9, kidnames: set("boy99", "girl1057", "girl877", "girl99"), aysalary: 127000, monthsummer: 3);
%p47254 := persistent Professor (id: 47254, name: "professorName47254", street: "xxxx_streetname", city: "city_name", state: "Pennsylvania", zipcode: "34306 ", picture: "picture", latitude: 1904, longitude: 1565, status: 0, aysalary: 127000, monthsummer: 1);
%p54453 := persistent Professor (id: 54453, name: "professorName54453", street: "xxxx_streetname", city: "city_name", state: "Virginia", zipcode: "23403 ", picture: "picture", latitude: 1111, longitude: 1753, status: 9, aysalary: 111000, monthsummer: 3);

// **Student file**

%ss65990 := persistent Student (id: 65990, name: "studentName65990", street: "xxxx_streetname", city: "city_name", state: "Vermont", zipcode: "24609", picture: "picture", latitude: 579, longitude: 534, studentno: 454356786);
%ss75001 := persistent Student (id: 75001, name: "studentName75001", street: "xxxx_streetname", city: "city_name", state: "Georgia", zipcode: "47880", picture: "picture", latitude: 1464, longitude: 179, studentno: 503582122);
%ss75051 := persistent Student (id: 75051, name: "studentName75051", street: "xxxx_streetname", city: "city_name", state: "Florida", zipcode: "63335", picture: "picture", latitude: 71, longitude: 755, studentno: 112339778);
%ss117525 := persistent Student (id: 117525, name: "studentName117525", street: "xxxx_streetname", city: "city_name", state: "Oklahoma", zipcode: "97695", picture: "picture", latitude: 1619, longitude: 900, studentno: 374361804);
%ss108981 := persistent Student (id: 108981, name: "studentName108981", street: "xxxx_streetname", city: "city_name", state: "West_Virginia", zipcode: "83589 ", picture: "picture", latitude: 1867, longitude: 985, studentno: 75613281); %ss47397 := persistent Student (id: 47397, name: "studentName47397", street: "xxxx_streetname", city: "city_name", state: "Iowa", zipcode: "64177 ", picture: "picture", latitude: 1043, longitude: 486, studentno: 651921317);
%ss71661 := persistent Student (id: 71661, name: "studentName71661", street: "xxxx_streetname", city: "city_name", state: "Idaho", zipcode: "64177 ", picture: "picture", latitude: 377, longitude: 859, studentno: 974503561);

// **TA file**

%ta65990 := persistent Ta (id: 65990, name: "studentName65990", street: "xxxx_streetname", city: "city_name", state: "Vermont", zipcode: "24609", picture: "picture", latitude: 579, longitude: 534, status: 6, semestersalary: 12000, apptfraction: 0.55);
%ta47397 := persistent Ta (id: 47397, name: "studentName47397", street: "xxxx_streetname", city: "city_name", state: "Iowa", zipcode: "64177 ", picture: "picture", latitude: 1043, longitude: 486,

status: 2, semestersalary: 16000, apptfraction: 0.45);
%ta71661 := persistent Ta (id: 71661, name: "studentName71661", street: "xxxx_streetname", city: "city_name", state: "Idaho", zipcode: "64177 ",picture: "picture", latitude: 377, longitude: 859, status: 9, semestersalary: 17000, apptfraction: 0.65);

// **Relationship definitions**

c220101->update()->sections.add(cc22010121);
c220101->update()->sections.add(cc22010122);
c10105->update()->sections.add(cc1010512);
c1112->update()->sections.add(cc111221);
c25121->update()->sections.add(cc2512122);
c220122->update()->sections.add(cc22012222);
c1126->update()->sections.add(cc112622);
c25135->update()->sections.add(cc2513512);
c220136->update()->sections.add(cc22013622);
p54453->update()->teaches.add(cc112622);
ta65990->update()->teaches.add(cc22012222);
p54453->update()->teaches.add(cc22013622);
p65805->update()->teaches.add(cc22010121);
ta65990->update()->teaches.add(cc22010122);
ta65990->update()->teaches.add(cc1010512);
p59247->update()->teaches.add(cc111221);
p54453->update()->teaches.add(cc2512122);
ta47397->update()->teaches.add(cc2513512);
%p59247.leads := d10;
%p65805.leads := d220;
%p47254.leads := d25;
%p54453.leads := d1;
cc22010121->update()->students.add(e10898122010121);
cc111221->update()->students.add(e75001111221);
cc1010512->update()->students.add(e750011010512);
cc22013622->update()->students.add(e4739722013622);
cc2512122->update()->students.add(e750512512122);
cc22010122->update()->students.add(e6599022010122);
cc22010122->update()->students.add(e11752522010122);
cc112622->update()->students.add(e71661112622);
cc22012222->update()->students.add(e7166122012222);
cc2513512->update()->students.add(e750512513512);
ss47397->update()->taken.add(e4739722013622);
ss65990->update()->taken.add(e6599022010122);
ss71661->update()->taken.add(e71661112622);
ss71661->update()->taken.add(e7166122012222);
ss75001->update()->taken.add(e75001111221);
ss75001->update()->taken.add(e750011010512);
ss75051->update()->taken.add(e750512512122);
ss75051->update()->taken.add(e750512513512);
ss108981->update()->taken.add(e10898122010121);
ss117525->update()->taken.add(e11752522010122);
d1->update()->offers.add(c1112);
d1->update()->offers.add(c1124);

```
d1->update()->offers.add(c1126);
d10->update()->offers.add(c10105);
d10->update()->offers.add(c10110);
d25->update()->offers.add(c25121);
d25->update()->offers.add(c25135);
d220->update()->offers.add(c220101);
d220->update()->offers.add(c220122);
d220->update()->offers.add(c220136);
d220->update()->employees.add(p59247);
d220->update()->employees.add(p65805);
d25->update()->employees.add(p47254);
d10->update()->employees.add(p54453);
d1->update()->employees.add(s1);
d1->update()->employees.add(s2);
d10->update()->employees.add(s3);
d220->update()->employees.add(ta65990);
d1->update()->employees.add(ta47397);
d25->update()->employees.add(ta71661);
d1->update()->students.add(ss65990);
d1->update()->students.add(ss75001);
d10->update()->students.add(ss75051);
d10->update()->students.add(ss117525);
d220->update()->students.add(ss108981);
d25->update()->students.add(ss47397);
d25->update()->students.add(ss71661);
p54453->update()->advises.add(ss65990);
p59247->update()->advises.add(ss75001);
p59247->update()->advises.add(ss75051);
p47254->update()->advises.add(ss117525);
p65805->update()->advises.add(ss108981);
p47254->update()->advises.add(ss47397);
p65805->update()->advises.add(ss71661);

%commit;

%begin;
%collect statistics;
%commit;
%cleanup;
};
```

C.2 ORDB of UniDB

C.2.1 ORDB Oracle 11_g schema

```
create type course_t
/
create type coursesection_t
/
```

```
create type department_t
/
create type employee_t
/
create type enrolled_t
/
create type instructor_t
/
create type person_t
/
create type professor_t
/
create type staff_t
/
create type student_t
/
create type ta_t
/
create type kids_t as object (kidname varchar2(8))
/
create type kids_ntt as table of kids_t
/
create or replace type course_ntt as table of ref course_t;
/
create or replace type coursesection_ntt as table of ref coursesection_t;
/
create or replace type enrolled_ntt as table of ref enrolled_t;
/
create or replace type employee_ntt as table of ref employee_t;
/
create or replace type student_ntt as table of ref student_t;
/
create or replace type course_t as object (
courseno number,
name varchar2(16),
credits number,
sections coursesection_ntt,
dept ref department_t) not final;
/
create or replace type coursesection_t as object (
sectionno number,
semester number,
textbook char(12),
nostudents number,
building char(8),
roomno number,
students enrolled_ntt,
course ref course_t,
teacher ref instructor_t) not final;
/
create or replace type department_t as object (
```

```
deptno number,
name varchar2(12),
building char(8),
budget number,
latitude number,
longitude number,
offers course_ntt,
employees employee_ntt,
students student_ntt,
chair ref professor_t) not final;
/
create or replace type enrolled_t as object (
grade varchar2(3),
section ref coursesection_t,
student ref student_t) not final;
/
create or replace type person_t as object (
id number,
name varchar2(20),
street char(15),
city char(9),
state varchar2(20),
zipcode char(9),
birthdate date,
picture char(7),
latitude number,
longitude number) not final;
/
create or replace type employee_t under person_t (
datehired date,
status number,
kidnames kids_ntt,
worksin ref department_t) not final;
/
create or replace type instructor_t under employee_t (
teaches coursesection_ntt) not final;
/
create or replace type professor_t under instructor_t (
aysalary number,
monthsummer number,
leads ref department_t,
advises student_ntt) final;
/
create or replace type staff_t under employee_t (
annualsalary number) final;
/
create or replace type student_t under person_t (
studentno number,
taken enrolled_ntt,
major ref department_t,
advisor ref professor_t) final;
```

```
/
create or replace type ta_t under instructor_t (
semestersalary number, apptfraction number) final;
/ create table course of course_t
nested table sections store as sections_nt
;
create table coursesection of coursesection_t
nested table students store as students_nt
;
create table department of department_t
nested table offers store as offers_nt
nested table employees store as employees_nt
nested table students store as students_nt1
;
create table enrolled of enrolled_t
;
create table professor of professor_t
nested table advises store as advises_nt
nested table kidnames store as kidnames_professor_nt
nested table teaches store as teaches_professor_nt
;
create table staff of staff_t
nested table kidnames store as kidnames_staff_nt
;
create table student of student_t
nested table taken store as taken_nt
;
create table ta of ta_t
nested table kidnames store as kidnames_ta_nt
nested table teaches store as teaches_ta_nt
;


alter table department add constraint dept_pk primary key (deptno);
alter table student add constraint stud_pk primary key (id);
alter table staff add constraint staff_pk primary key (id);
alter table professor add constraint prof_pk primary key (id);
alter table ta add constraint ta_pk primary key (id);

alter type course_t add attribute (uoid char (25)) cascade;
alter type coursesection_t add attribute (uoid char (25)) cascade;
alter type department_t add attribute (uoid char (25)) cascade;
alter type enrolled_t add attribute (uoid char (25)) cascade;
alter type person_t add attribute (uoid char (25)) cascade;

alter table department add (scope for (chair) is professor);
alter table enrolled add (scope for (section) is coursesection);
alter table enrolled add (scope for (student) is student);
alter table staff add (scope for (worksin) is department);
alter table ta add (scope for (worksin) is department);
alter table professor add (scope for (worksin) is department);
alter table professor add (scope for (leads) is department);
alter table student add (scope for (major) is department);
```

```
alter table student add (scope for (advisor) is professor);
alter table course add (scope for (dept) is department);
alter table coursesection add (scope for (course) is course);
```

C.2.2 Functions for ORDB UniDB

```
alter type employee_t add member function salary return number
deterministic cascade;
create or replace type body employee_t as
member function salary return number is
begin
   return 0;
end;
end;
/
alter type instructor_t add overriding member function salary return number
deterministic cascade;
create or replace type body instructor_t as
overriding member function salary return number is
begin
   return 0;
end;
end;
/
alter type professor_t add overriding member function salary return number
deterministic cascade;
create or replace type body professor_t as
overriding member function salary return number is
begin
   return (aysalary * (9 + monthsummer) / 9.0 );
end;
end;
/
alter type staff_t add overriding member function salary return number
deterministic cascade;
create or replace type body staff_t as
overriding member function salary return number is
begin
   return annualsalary;
end;
end;
/
alter type ta_t add overriding member function salary return number
deterministic cascade;
create or replace type body ta_t as
overriding member function salary return number is
begin
   return ( apptfraction * ( 2 * semestersalary ));
end;
end;
```

/

C.2.3 ORDB UniDB data

// **Object definition files**

// **Course file**

insert into course values (124, 'coursename124', 1, coursesection_ntt(), null, '1124');
insert into course values (126, 'coursename126', 4, coursesection_ntt(), null, '1126');
insert into course values (112, 'coursename112', 1, coursesection_ntt(), null, '1112');
insert into course values (122, 'coursename122', 2, coursesection_ntt(), null, '220122');
insert into course values (136, 'coursename136', 3, coursesection_ntt(), null, '220136');
insert into course values (101, 'coursename101', 4, coursesection_ntt(), null, '220101');
insert into course values (110, 'coursename110', 3, coursesection_ntt(), null, '10110');
insert into course values (105, 'coursename105', 4, coursesection_ntt(), null, '10105');
insert into course values (121, 'coursename121', 4, coursesection_ntt(), null, '25121');
insert into course values (135, 'coursename135', 3, coursesection_ntt(), null, '25135');

// **Coursesection file**

insert into coursesection values (2, 2, 'textbookname', 20, 'building', 39, enrolled_ntt(), null, null, '112622');
insert into coursesection values (2, 2, 'textbookname', 20, 'building', 70, enrolled_ntt(), null, null, '22012222');
insert into coursesection values (2, 2, 'textbookname', 20, 'building', 91, enrolled_ntt(), null, null, '22013622');
insert into coursesection values (2, 1, 'textbookname', 20, 'building', 50, enrolled_ntt(), null, null, '22010121');
insert into coursesection values (2, 2, 'textbookname', 20, 'building', 50, enrolled_ntt(), null, null, '22010122');
insert into coursesection values (1, 2, 'textbookname', 20, 'building', 75, enrolled_ntt(), null, null, '1010512');
insert into coursesection values (2, 1, 'textbookname', 20, 'building', 10, enrolled_ntt(), null, null, '111221');
insert into coursesection values (1, 2, 'textbookname', 20, 'building', 13, enrolled_ntt(), null, null, '2513512');
insert into coursesection values (2, 2, 'textbookname', 20, 'building', 43, enrolled_ntt(), null, null, '2512122');

// **Department file**

insert into department values (1, 'deptname1', 'building', 6000000, 47, 6, course_ntt(), employee_ntt(), student_ntt(), null, '1');
insert into department values (10, 'deptname10', 'building', 5000000, 74, 38, course_ntt(), employee_ntt(), student_ntt(), null, '10');
insert into department values (220, 'deptname220', 'building', 6000000, 23, 37, course_ntt(), employee_ntt(), student_ntt(), null, '220');
insert into department values (25, 'deptname25', 'building', 8000000, 55, 12, course_ntt(), employee_ntt(), student_ntt(), null, '25');

// **Enrolled file**

insert into enrolled values ('AB', null, null, '71661112622');
insert into enrolled values ('F', null, null, '7166122012222');
insert into enrolled values ('D', null, null, '4739722013622');
insert into enrolled values ('AB', null, null, '6599022010122');
insert into enrolled values ('A', null, null, '750011010512');
insert into enrolled values ('AB', null, null, '75001111221');
insert into enrolled values ('C', null, null, '750512512122');
insert into enrolled values ('B', null, null, '750512513512');
insert into enrolled values ('A', null, null, '11752522010122');
insert into enrolled values ('AB', null, null, '10898122010121');

// **Professor file**

insert into professor values (professor_t(59247, 'professorName59247', 'xxxx_streetname', 'city_name', 'Utah', '98075', '1971-03-08', 'picture', 1211, 98, '59247', '1958-08-08', 9, kids_ntt(kids_t('boy90'), kids_t('girl39')), null, coursesection_ntt(), 115000, 2, null, student_ntt()));
insert into professor values (professor_t(65805, 'professorName65805', 'xxxx_streetname', 'city_name', 'Minnesota', '92344', '1983-10-30', 'picture', 1337, 60, '65805', '1943-04-19', 9, kids_ntt(kids_t('boy99'), kids_t('girl1057'), kids_t('girl877'), kids_t('girl99')), null, coursesection_ntt(), 127000, 3, null, student_ntt()));
insert into professor values (professor_t(47254, 'professorName47254', 'xxxx_streetname', 'city_name', 'Pennsylvania', '34306', '1981-12-03 00:00:00', 'picture', 1904, 1565, '47254', '1982-01-09', 0, kids_ntt(), null, coursesection_ntt(), 127000, 1, null, student_ntt()));
insert into professor values (professor_t(54453, 'professorName54453', 'xxxx_streetname', 'city_name', 'Virginia', '23403', '1971-04-04', 'picture', 1111, 1753, '54453', '1970-05-14', 9, kids_ntt(), null, coursesection_ntt(), 111000, 3, null, student_ntt()));

// **Staff file**

insert into staff values (staff_t(1, 'staffName1', 'xxxx_streetname', 'city_name', 'Oklahoma', '41421', '1989-11-13', 'picture', 362, 27, '1', '1955-11-28', 3, kids_ntt(kids_t('boy90'), kids_t('girl90')), null, 83000));
insert into staff values (staff_t(2, 'staffName2', 'xxxx_streetname', 'city_name', 'Oregon', '56429', '1957-12-09', 'picture', 1782, 1530, '2', '1966-11-01', 2, kids_ntt(kids_t('boy62'), kids_t('girl62')), null, 33000));
insert into staff values (staff_t(3, 'staffName3', 'xxxx_streetname', 'city_name', 'Florida', '18456', '1983-06-28', 'picture', 1011, 42, '3', '1942-08-30', 2, kids_ntt(kids_t('boy90'), kids_t('girl29')), null, 47000));

// **Student file**

insert into student values (student_t(65990, 'studentName65990', 'xxxx_streetname', 'city_name', 'Vermont', '24609', '1941-04-16', 'picture', 579, 534, '65990', 454356786, enrolled_ntt(), null, null));
insert into student values (student_t(75001, 'studentName75001', 'xxxx_streetname', 'city_name', 'Georgia', '47880', '1985-07-21', 'picture', 1464, 179, '75001', 503582122, enrolled_ntt(), null, null));
insert into student values (student_t(75051, 'studentName75051', 'xxxx_streetname', 'city_name', 'Florida', '63335', '1956-12-18', 'picture', 71, 755, '75051', 112339778, enrolled_ntt(), null, null));
insert into student values (student_t(117525, 'studentName117525', 'xxxx_streetname', 'city_name', 'Oklahoma', '97695', '1952-08-27', 'picture', 1619, 900, '117525', 374361804, enrolled_ntt(), null, null));

insert into student values (student_t(108981, 'studentName108981', 'xxxx_streetname', 'city_name', 'West_Virginia', '83589', '1965-09-02', 'picture', 1867, 985, '108981', 75613281, enrolled_ntt(), null, null));
insert into student values (student_t(47397, 'studentName47397', 'xxxx_streetname', 'city_name', 'Iowa', '64177', '1970-10-12', 'picture', 1043, 486, '47397', 651921317, enrolled_ntt(), null, null));
insert into student values (student_t(71661, 'studentName71661', 'xxxx_streetname', 'city_name', 'Idaho', '64177', '1952-04-17', 'picture', 377, 859, '71661', 974503561, enrolled_ntt(), null, null));

// **TA file**

insert into ta values (ta_t(65990, 'studentName65990', 'xxxx_streetname', 'city_name', 'Vermont', '24609', '1941-04-16', 'picture', 579, 534, '65990', '1987-09-29', 6, kids_ntt(), null, coursesection_ntt(), 12000, .55));
insert into ta values (ta_t(47397, 'studentName47397', 'xxxx_streetname', 'city_name', 'Iowa', '64177', '1970-10-12', 'picture', 1043, 486, '47397', '1963-11-27', 2, kids_ntt(), null, coursesection_ntt(), 16000, .45));
insert into ta values (ta_t(71661, 'studentName71661', 'xxxx_streetname', 'city_name', 'Idaho', '64177', '1952-04-17', 'picture', 377, 859, '71661', '1963-11-27', 9, kids_ntt(), null, coursesection_ntt(), 17000, .65));

// **Relationship definition files**

// **Course_Relationships file**

insert into table (select sections from course where course.uoid = '220101') select ref(c) from coursesection c where c.uoid in ('22010121', '22010122');
insert into table (select sections from course where course.uoid = '10105') select ref(c) from coursesection c where c.uoid in ('1010512');
insert into table (select sections from course where course.uoid = '1112') select ref(c) from coursesection c where c.uoid in ('111221');
insert into table (select sections from course where course.uoid = '25121') select ref(c) from coursesection c where c.uoid in ('2512122');
insert into table (select sections from course where course.uoid = '220122') select ref(c) from coursesection c where c.uoid in ('22012222');
insert into table (select sections from course where course.uoid = '1126') select ref(c) from coursesection c where c.uoid in ('112622');
insert into table (select sections from course where course.uoid = '25135') select ref(c) from coursesection c where c.uoid in ('2513512');
insert into table (select sections from course where course.uoid = '220136') select ref(c) from coursesection c where c.uoid in ('22013622');
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where course.uoid = '1124';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where course.uoid = '1126';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where course.uoid = '1112';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where course.uoid = '220122';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where course.uoid = '220136';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid

= '220') where course.uoid = '220101';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where course.uoid = '10110';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where course.uoid = '10105';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where course.uoid = '25121';
update course set dept = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where course.uoid = '25135';

// **Coursesection_Relationships file**

insert into table (select students from coursesection where coursesection.uoid = '22010121') select ref(e) from enrolled e where e.uoid in ('10898122010121');
insert into table (select students from coursesection where coursesection.uoid = '111221') select ref(e) from enrolled e where e.uoid in ('75001111221');
insert into table (select students from coursesection where coursesection.uoid = '1010512') select ref(e) from enrolled e where e.uoid in ('750011010512');
insert into table (select students from coursesection where coursesection.uoid = '22013622') select ref(e) from enrolled e where e.uoid in ('4739722013622');
insert into table (select students from coursesection where coursesection.uoid = '2512122') select ref(e) from enrolled e where e.uoid in ('750512512122');
insert into table (select students from coursesection where coursesection.uoid = '22010122') select ref(e) from enrolled e where e.uoid in ('6599022010122', '11752522010122');
insert into table (select students from coursesection where coursesection.uoid = '112622') select ref(e) from enrolled e where e.uoid in ('71661112622');
insert into table (select students from coursesection where coursesection.uoid = '22012222') select ref(e) from enrolled e where e.uoid in ('7166122012222');
insert into table (select students from coursesection where coursesection.uoid = '2513512') select ref(e) from enrolled e where e.uoid in ('750512513512');
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '220101') where coursesection.uoid = '22010121';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '220101') where coursesection.uoid = '22010122';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '10105') where coursesection.uoid = '1010512';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '1112') where coursesection.uoid = '111221';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '25121') where coursesection.uoid = '2512122';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '220122') where coursesection.uoid = '22012222';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '1126') where coursesection.uoid = '112622';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '25135') where coursesection.uoid = '2513512';
update coursesection set course = (select treat (ref(c) as ref course_t) from course c where c.uoid = '220136') where coursesection.uoid = '22013622';
update coursesection set teacher = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '54453') where coursesection.uoid = '112622';
update coursesection set teacher = (select treat (ref(t) as ref ta_t) from ta t where t.uoid = '65990')

where coursesection.uoid = '22012222';
update coursesection set teacher = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '54453') where coursesection.uoid = '22013622';
update coursesection set teacher = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '65805') where coursesection.uoid = '22010121';
update coursesection set teacher = (select treat (ref(t) as ref ta_t) from ta t where t.uoid = '65990') where coursesection.uoid = '22010122';
update coursesection set teacher = (select treat (ref(t) as ref ta_t) from ta t where t.uoid = '65990') where coursesection.uoid = '1010512';
update coursesection set teacher = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '59247') where coursesection.uoid = '111221';
update coursesection set teacher = (select treat (ref(t) as ref ta_t) from ta t where t.uoid = '47397') where coursesection.uoid = '2513512';
update coursesection set teacher = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '54453') where coursesection.uoid = '2512122';

// **Department_Relationships file**

insert into table (select offers from department where department.uoid = '1') select ref(c) from course c where c.uoid in ('1112', '1124', '1126');
insert into table (select offers from department where department.uoid = '10') select ref(c) from course c where c.uoid in ('10105', '10110');
insert into table (select offers from department where department.uoid = '25') select ref(c) from course c where c.uoid in ('25121', '25135');
insert into table (select offers from department where department.uoid = '220') select ref(c) from course c where c.uoid in ('220101', '220122', '220136');
insert into table (select employees from department where department.uoid = '1') select ref(s) from staff s where s.uoid in ('1', '2');
insert into table (select employees from department where department.uoid = '1') select ref(t) from ta t where t.uoid in ('47397');
insert into table (select employees from department where department.uoid = '10') select ref(s) from staff s where s.uoid in ('3');
insert into table (select employees from department where department.uoid = '10') select ref(p) from professor p where p.uoid in ('54453');
insert into table (select employees from department where department.uoid = '25') select ref(p) from professor p where p.uoid in ('47254');
insert into table (select employees from department where department.uoid = '25') select ref(t) from ta t where t.uoid in ('71661');
insert into table (select employees from department where department.uoid = '220') select ref(p) from professor p where p.uoid in ('59247', '65805');
insert into table (select employees from department where department.uoid = '220') select ref(t) from ta t where t.uoid in ('65990');
insert into table (select students from department where department.uoid = '1') select ref(s) from student s where s.uoid in ('65990', '75001');
insert into table (select students from department where department.uoid = '10') select ref(s) from student s where s.uoid in ('75051', '117525');
insert into table (select students from department where department.uoid = '25') select ref(s) from student s where s.uoid in ('47397', '71661');
insert into table (select students from department where department.uoid = '220') select ref(s) from student s where s.uoid in ('108981');
update department set chair = (select treat (ref(p) as ref professor_t) from professor p where p.uoid

= '54453') where department.uoid = '1';
update department set chair = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '59247') where department.uoid = '10';
update department set chair = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '65805') where department.uoid = '220';
update department set chair = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '47254') where department.uoid = '25';

// **Enrolled_Relationships file**

update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '22010121') where enrolled.uoid = '10898122010121';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '111221') where enrolled.uoid = '75001111221';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '1010512') where enrolled.uoid = '750011010512';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '2513512') where enrolled.uoid = '750512513512';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '22013622') where enrolled.uoid = '4739722013622';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '22010122') where enrolled.uoid = '6599022010122';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '22012222') where enrolled.uoid = '7166122012222';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '112622') where enrolled.uoid = '71661112622';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '2512122') where enrolled.uoid = '750512512122';
update enrolled set section = (select treat (ref(c) as ref coursesection_t) from coursesection c where c.uoid = '22010122') where enrolled.uoid = '11752522010122';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '71661') where enrolled.uoid = '71661112622';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '71661') where enrolled.uoid = '7166122012222';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '47397') where enrolled.uoid = '4739722013622';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '65990') where enrolled.uoid = '6599022010122';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '75001') where enrolled.uoid = '750011010512';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '75001') where enrolled.uoid = '75001111221';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '75051') where enrolled.uoid = '750512512122';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '75051') where enrolled.uoid = '750512513512';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '117525') where enrolled.uoid = '11752522010122';
update enrolled set student = (select treat (ref(s) as ref student_t) from student s where s.uoid = '108981') where enrolled.uoid = '10898122010121';

// **Professor_Relationships file**

update professor set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where professor.uoid = '59247';
update professor set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where professor.uoid = '65805';
update professor set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where professor.uoid = '47254';
update professor set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where professor.uoid = '54453';
insert into table (select teaches from professor where professor.uoid = '54453') select ref(c) from coursesection c where c.uoid in ('112622', '22013622', '2512122');
insert into table (select teaches from professor where professor.uoid = '59247') select ref(c) from coursesection c where c.uoid in ('111221');
insert into table (select teaches from professor where professor.uoid = '65805') select ref(c) from coursesection c where c.uoid in ('22010121');
update professor set leads = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where professor.uoid = '59247';
update professor set leads = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where professor.uoid = '65805';
update professor set leads = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where professor.uoid = '47254';
update professor set leads = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where professor.uoid = '54453';
insert into table (select advises from professor where professor.uoid = '47254') select ref(s) from student s where s.uoid in ('117525', '47397');
insert into table (select advises from professor where professor.uoid = '54453') select ref(s) from student s where s.uoid in ('65990');
insert into table (select advises from professor where professor.uoid = '59247') select ref(s) from student s where s.uoid in ('75001', '75051');
insert into table (select advises from professor where professor.uoid = '65805') select ref(s) from student s where s.uoid in ('108981', '71661');

// **Staff_Relationships file**

update staff set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where staff.uoid = '1';
update staff set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where staff.uoid = '2';
update staff set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where staff.uoid = '3';

// **Student_Relationships file**

insert into table (select taken from student where student.uoid = '47397') select ref(e) from enrolled e where e.uoid in ('4739722013622');
insert into table (select taken from student where student.uoid = '65990') select ref(e) from enrolled e where e.uoid in ('6599022010122');
insert into table (select taken from student where student.uoid = '71661') select ref(e) from enrolled e where e.uoid in ('71661112622', '7166122012222');
insert into table (select taken from student where student.uoid = '75001') select ref(e) from enrolled e where e.uoid in ('75001111221', '750011010512');

insert into table (select taken from student where student.uoid = '75051') select ref(e) from enrolled e where e.uoid in ('750512512122', '750512513512');
insert into table (select taken from student where student.uoid = '108981') select ref(e) from enrolled e where e.uoid in ('10898122010121');
insert into table (select taken from student where student.uoid = '117525') select ref(e) from enrolled e where e.uoid in ('11752522010122');
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where student.uoid = '65990';
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where student.uoid = '75001';
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where student.uoid = '75051';
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '10') where student.uoid = '117525';
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where student.uoid = '108981';
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where student.uoid = '47397';
update student set major = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where student.uoid = '71661';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '54453') where student.uoid = '65990';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '59247') where student.uoid = '75001';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '59247') where student.uoid = '75051';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '47254') where student.uoid = '117525';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '65805') where student.uoid = '108981';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '47254') where student.uoid = '47397';
update student set advisor = (select treat (ref(p) as ref professor_t) from professor p where p.uoid = '65805') where student.uoid = '71661';

// **Worksin_Relationships file**

update ta set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '220') where ta.uoid = '65990';
update ta set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '1') where ta.uoid = '47397';
update ta set worksin = (select treat (ref(d) as ref department_t) from department d where d.uoid = '25') where ta.uoid = '71661';
insert into table (select teaches from ta where ta.uoid = '47397') select ref(c) from coursesection c where c.uoid in ('2513512');
insert into table (select teaches from ta where ta.uoid = '65990') select ref(c) from coursesection c where c.uoid in ('22012222', '22010122', '1010512');

C.3 XML Schema documents of UniDB

C.3.1 Schema document

```
<?xml version="1.0" encoding="UTF-8"? >
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:annotation>
<xs:documentation xml:lang ="en">
Generated XML Schema </xs:documentation>
</xs:annotation>
<xs:element name= "XMLSchema">
<xs:complexType>
   <xs:sequence>
      <xs:element name = "course" type= "course_t" maxOccurs = "unbounded"/>
      <xs:element name = "coursesection" type= "coursesection_t"
maxOccurs = "unbounded"/>
      <xs:element name = "department" type= "department_t" maxOccurs = "unbounded"/>
      <xs:element name = "enrolled" type= "enrolled_t" maxOccurs = "unbounded"/>
      <xs:element name = "professor" type= "professor_t" maxOccurs = "unbounded"/>
      <xs:element name = "staff" type= "staff_t" maxOccurs = "unbounded"/>
      <xs:element name = "student" type= "student_t" maxOccurs = "unbounded"/>
      <xs:element name = "ta" type= "ta_t" maxOccurs = "unbounded"/>
   </xs:sequence>
</xs:complexType>
<xs:key name= "coursedeptnocoursenoPK">
   <xs:selector xpath= ".//course"/>
   <xs:field xpath= "deptno"/>
   <xs:field xpath= "courseno"/>
</xs:key>
<xs:key name= "coursesectiondeptnocoursenosectionnosemesterPK">
   <xs:selector xpath= ".//coursesection"/>
   <xs:field xpath= "deptno"/>
   <xs:field xpath= "courseno"/>
   <xs:field xpath= "sectionno"/>
   <xs:field xpath= "semester"/>
</xs:key>
<xs:key name= "departmentdeptnoPK">
   <xs:selector xpath= ".//department"/>
   <xs:field xpath= "deptno"/>
</xs:key>
<xs:key name= "enrolledstudentiddeptnocoursenosectionnosemesterPK">
   <xs:selector xpath= ".//enrolled"/>
   <xs:field xpath= "studentid"/>
   <xs:field xpath= "deptno"/>
   <xs:field xpath= "courseno"/>
   <xs:field xpath= "sectionno"/>
   <xs:field xpath= "semester"/>
</xs:key>
<xs:key name= "professoridPK">
   <xs:selector xpath= ".//professor"/>
```

```
   <xs:field xpath= "id"/>
</xs:key>
<xs:key name= "staffidPK">
   <xs:selector xpath= ".//staff"/>
      <xs:field xpath= "id"/>
</xs:key>
<xs:key name= "studentidPK">
   <xs:selector xpath= ".//student"/>
      <xs:field xpath= "id"/>
</xs:key>
<xs:key name= "taidPK">
   <xs:selector xpath= ".//ta"/>
      <xs:field xpath= "id"/>
</xs:key>
<xs:unique name= "departmentchairUK">
   <xs:selector xpath= ".//department"/>
      <xs:field xpath= "chair"/>
</xs:unique>
<xs:unique name= "departmentnameUK">
   <xs:selector xpath= ".//department"/>
      <xs:field xpath= "name"/>
</xs:unique>
<xs:keyref name= "coursedeptnoFK" refer= "departmentdeptnoPK">
   <xs:selector xpath= ".//course"/>
      <xs:field xpath= "deptno"/>
</xs:keyref>
<xs:keyref name= "coursesectiondeptnocoursenoFK" refer= "coursedeptnocoursenoPK">
   <xs:selector xpath= ".//coursesection"/>
      <xs:field xpath= "deptno"/>
      <xs:field xpath= "courseno"/>
</xs:keyref>
<xs:keyref name= "departmentchairFK" refer= "professoridPK">
   <xs:selector xpath= ".//department"/>
      <xs:field xpath= "chair"/>
</xs:keyref>
<xs:keyref name= "enrolleddeptnocoursenosectionnosemesterFK"
refer= "coursesectiondeptnocoursenosectionnosemesterPK">
   <xs:selector xpath= ".//enrolled"/>
      <xs:field xpath= "deptno"/>
      <xs:field xpath= "courseno"/>
      <xs:field xpath= "sectionno"/>
      <xs:field xpath= "semester"/>
</xs:keyref>
<xs:keyref name= "enrolledstudentidFK" refer= "studentidPK">
   <xs:selector xpath= ".//enrolled"/>
      <xs:field xpath= "studentid"/>
</xs:keyref>
<xs:keyref name= "studentmajordeptFK" refer= "departmentdeptnoPK">
   <xs:selector xpath= ".//student"/>
      <xs:field xpath= "majordept"/>
</xs:keyref>
```

```
<xs:keyref name= "studentadvisorFK" refer= "professoridPK">
   <xs:selector xpath= ".//student"/>
      <xs:field xpath= "advisor"/>
</xs:keyref>
</xs:element>
<xs:complexType name = "course_t">
   <xs:sequence>
      <xs:element name = "deptno" type= "xs:decimal"/>
      <xs:element name = "courseno" type= "xs:decimal"/>
      <xs:element name = "name" type= "xs:string"/>
      <xs:element name = "credits" type= "xs:decimal" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "coursesection_t">
   <xs:sequence>
      <xs:element name = "deptno" type= "xs:decimal"/>
      <xs:element name = "courseno" type= "xs:decimal"/>
      <xs:element name = "sectionno" type= "xs:decimal"/>
      <xs:element name = "instructorid" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "semester" type= "xs:decimal"/>
      <xs:element name = "textbook" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "nostudents" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "building" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "roomno" type= "xs:decimal" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "department_t">
   <xs:sequence>
      <xs:element name = "deptno" type= "xs:decimal"/>
      <xs:element name = "name" type= "xs:string"/>
      <xs:element name = "building" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "budget" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "chair" type= "xs:decimal"/>
      <xs:element name = "latitude" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "longitude" type= "xs:decimal" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "employee_t">
   <xs:complexContent>
   <xs:extension base= "person_t">
      <xs:sequence>
      <xs:element name = "dept" type= "xs:decimal"/>
      <xs:element name = "datehired" type= "xs:date" minOccurs= "0"/>
      <xs:element name = "status" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "kidnames" type= "xs:string" minOccurs= "0"
maxOccurs= "unbounded"/>
      </xs:sequence>
   </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name = "enrolled_t">
```

```
   <xs:sequence>
      <xs:element name = "studentid" type= "xs:decimal"/>
      <xs:element name = "deptno" type= "xs:decimal"/>
      <xs:element name = "courseno" type= "xs:decimal"/>
      <xs:element name = "sectionno" type= "xs:decimal"/>
      <xs:element name = "semester" type= "xs:decimal"/>
      <xs:element name = "grade" type= "xs:string" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "instructor_t">
   <xs:complexContent>
   <xs:extension base= "employee_t">
      <xs:sequence>
      </xs:sequence>
   </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name = "person_t">
   <xs:sequence>
      <xs:element name = "id" type= "xs:decimal"/>
      <xs:element name = "name" type= "xs:string"/>
      <xs:element name = "street" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "city" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "state" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "zipcode" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "birthdate" type= "xs:date" minOccurs= "0"/>
      <xs:element name = "picture" type= "xs:string" minOccurs= "0"/>
      <xs:element name = "latitude" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "longitude" type= "xs:decimal" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "professor_t">
   <xs:complexContent>
   <xs:extension base= "instructor_t">
      <xs:sequence>
      <xs:element name = "aysalary" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "monthsummer" type= "xs:decimal" minOccurs= "0"/>
      </xs:sequence>
   </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name = "staff_t">
   <xs:complexContent>
   <xs:extension base= "employee_t">
      <xs:sequence>
      <xs:element name = "annualsalary" type= "xs:decimal" minOccurs= "0"/>
      </xs:sequence>
   </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name = "student_t">
```

```
   <xs:complexContent>
   <xs:extension base= "person_t">
      <xs:sequence>
      <xs:element name = "studentno" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "majordept" type= "xs:decimal"/>
      <xs:element name = "advisor" type= "xs:decimal"/>
      </xs:sequence>
   </xs:extension>
   </xs:complexContent>
</xs:complexType>
<xs:complexType name = "ta_t">
   <xs:complexContent>
   <xs:extension base= "instructor_t">
      <xs:sequence>
      <xs:element name = "semestersalary" type= "xs:decimal" minOccurs= "0"/>
      <xs:element name = "apptfraction" type= "xs:decimal" minOccurs= "0"/>
      </xs:sequence>
   </xs:extension>
   </xs:complexContent>
</xs:complexType>
</xs:schema>
```

C.3.2 Instance document

```
<?xml version="1.0" encoding="UTF-8"? >
<XMLSchema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi: noNamespaceSchemaLocation="XMLSchema.xsd" >
  <course>
      <deptno>1</deptno>
     <courseno>124</courseno>
     <name>coursename124</name>
     <credits>1</credits>
  </course>
  <course>
     <deptno>1</deptno>
     <courseno>126</courseno>
     <name>coursename126</name>
     <credits>4</credits>
  </course>
  <course>
     <deptno>1</deptno>
     <courseno>112</courseno>
     <name>coursename112</name>
     <credits>1</credits>
  </course>
  <course>
     <deptno>220</deptno>
     <courseno>122</courseno>
     <name>coursename122</name>
     <credits>2</credits>
```

```
</course>
<course>
   <deptno>220</deptno>
   <courseno>136</courseno>
   <name>coursename136</name>
   <credits>3</credits>
</course>
<course>
  <deptno>220</deptno>
  <courseno>101</courseno>
  <name>coursename101</name>
  <credits>4</credits>
</course>
<course>
  <deptno>10</deptno>
  <courseno>110</courseno>
  <name>coursename110</name>
  <credits>3</credits>
</course>
<course>
  <deptno>10</deptno>
  <courseno>105</courseno>
  <name>coursename105</name>
  <credits>4</credits>
</course>
<course>
  <deptno>25</deptno>
  <courseno>121</courseno>
  <name>coursename121</name>
  <credits>4</credits>
</course>
<course>
  <deptno>25</deptno>
  <courseno>135</courseno>
  <name>coursename135</name>
  <credits>3</credits>
</course>
<coursesection>
  <deptno>1</deptno>
  <courseno>126</courseno>
  <sectionno>2</sectionno>
  <instructorid>54453</instructorid>
  <semester>2</semester>
  <textbook>textbookname</textbook>
  <nostudents>20</nostudents>
  <building>building</building>
  <roomno>39</roomno>
</coursesection>
<coursesection>
  <deptno>220</deptno>
  <courseno>122</courseno>
```

```
    <sectionno>2</sectionno>
    <instructorid>65990</instructorid>
    <semester>2</semester>
    <textbook>textbookname</textbook>
    <nostudents>20</nostudents>
    <building>building</building>
    <roomno>70</roomno>
  </coursesection>
  <coursesection>
    <deptno>220</deptno>
    <courseno>136</courseno>
    <sectionno>2</sectionno>
    <instructorid>54453</instructorid>
    <semester>2</semester>
    <textbook>textbookname</textbook>
    <nostudents>20</nostudents>
    <building>building</building>
    <roomno>91</roomno>
  </coursesection>
  <coursesection>
    <deptno>220</deptno>
    <courseno>101</courseno>
    <sectionno>2</sectionno>
    <instructorid>65805</instructorid>
    <semester>1</semester>
    <textbook>textbookname</textbook>
    <nostudents>20</nostudents>
    <building>building</building>
    <roomno>50</roomno>
  </coursesection>
  <coursesection>
    <deptno>220</deptno>
    <courseno>101</courseno>
    <sectionno>2</sectionno>
    <instructorid>65990</instructorid>
    <semester>2</semester>
    <textbook>textbookname</textbook>
    <nostudents>20</nostudents>
    <building>building</building>
    <roomno>50</roomno>
  </coursesection>
  <coursesection>
    <deptno>10</deptno>
    <courseno>105</courseno>
    <sectionno>1</sectionno>
    <instructorid>65990</instructorid>
    <semester>2</semester>
    <textbook>textbookname</textbook>
    <nostudents>20</nostudents>
    <building>building</building>
    <roomno>75</roomno>
```

```
</coursesection>
<coursesection>
  <deptno>1</deptno>
  <courseno>112</courseno>
  <sectionno>2</sectionno>
  <instructorid>59247</instructorid>
  <semester>1</semester>
  <textbook>textbookname</textbook>
  <nostudents>20</nostudents>
  <building>building</building>
  <roomno>10</roomno>
</coursesection>
<coursesection>
  <deptno>25</deptno>
  <courseno>135</courseno>
  <sectionno>1</sectionno>
  <instructorid>47397</instructorid>
  <semester>2</semester>
  <textbook>textbookname</textbook>
  <nostudents>20</nostudents>
  <building>building</building>
  <roomno>13</roomno>
</coursesection>
<coursesection>
  <deptno>25</deptno>
  <courseno>121</courseno>
  <sectionno>2</sectionno>
  <instructorid>54453</instructorid>
  <semester>2</semester>
  <textbook>textbookname</textbook>
  <nostudents>20</nostudents>
  <building>building</building>
  <roomno>43</roomno>
</coursesection>
<department>
  <deptno>1</deptno>
  <name>deptname1</name>
  <building>building</building>
  <budget>6000000</budget>
  <chair>54453</chair>
  <latitude>47</latitude>
  <longitude>6</longitude>
</department>
<department>
  <deptno>10</deptno>
  <name>deptname10</name>
  <building>building</building>
  <budget>5000000</budget>
  <chair>59247</chair>
  <latitude>74</latitude>
  <longitude>38</longitude>
```

```
</department>
<department>
  <deptno>220</deptno>
  <name>deptname220</name>
  <building>building</building>
  <budget>6000000</budget>
  <chair>65805</chair>
  <latitude>23</latitude>
  <longitude>37</longitude>
</department>
<department>
  <deptno>25</deptno>
  <name>deptname25</name>
  <building>building</building>
  <budget>8000000</budget>
  <chair>47254</chair>
  <latitude>55</latitude>
  <longitude>12</longitude>
</department>
<enrolled>
  <studentid>71661</studentid>
  <deptno>1</deptno>
  <courseno>126</courseno>
  <sectionno>2</sectionno>
  <semester>2</semester>
  <grade>AB</grade>
</enrolled>
<enrolled>
  <studentid>71661</studentid>
  <deptno>220</deptno>
  <courseno>122</courseno>
  <sectionno>2</sectionno>
  <semester>2</semester>
  <grade>F</grade>
</enrolled>
<enrolled>
  <studentid>47397</studentid>
  <deptno>220</deptno>
  <courseno>136</courseno>
  <sectionno>2</sectionno>
  <semester>2</semester>
  <grade>D</grade>
</enrolled>
<enrolled>
  <studentid>65990</studentid>
  <deptno>220</deptno>
  <courseno>101</courseno>
  <sectionno>2</sectionno>
  <semester>2</semester>
  <grade>AB</grade>
</enrolled>
```

```
<enrolled>
  <studentid>75001</studentid>
  <deptno>10</deptno>
  <courseno>105</courseno>
  <sectionno>1</sectionno>
  <semester>2</semester>
  <grade>A</grade>
</enrolled>
<enrolled>
  <studentid>75001</studentid>
  <deptno>1</deptno>
  <courseno>112</courseno>
  <sectionno>2</sectionno>
  <semester>1</semester>
  <grade>AB</grade>
</enrolled>
<enrolled>
  <studentid>75051</studentid>
  <deptno>25</deptno>
  <courseno>121</courseno>
  <sectionno>2</sectionno>
  <semester>2</semester>
  <grade>C</grade>
</enrolled>
<enrolled>
  <studentid>75051</studentid>
  <deptno>25</deptno>
  <courseno>135</courseno>
  <sectionno>1</sectionno>
  <semester>2</semester>
  <grade>B</grade>
</enrolled>
<enrolled>
  <studentid>117525</studentid>
  <deptno>220</deptno>
  <courseno>101</courseno>
  <sectionno>2</sectionno>
  <semester>2</semester>
  <grade>A</grade>
</enrolled>
<enrolled>
  <studentid>108981</studentid>
  <deptno>220</deptno>
  <courseno>101</courseno>
  <sectionno>2</sectionno>
  <semester>1</semester>
  <grade>AB</grade>
</enrolled>
<professor>
  <id>59247</id>
  <name>professorName59247</name>
```

```
      <street>xxxx_streetname</street>
      <city>city_name</city>
      <state>Utah</state>
      <zipcode>98075 </zipcode>
      <birthdate>1971-03-08</birthdate>
      <picture>picture</picture>
      <latitude>1211</latitude>
      <longitude>98</longitude>
      <dept>220</dept>
      <datehired>1958-08-08</datehired>
      <status>9</status>
      <kidnames>boy90</kidnames>
      <kidnames>girl39</kidnames>
      <aysalary>115000</aysalary>
      <monthsummer>2</monthsummer>
</professor>
  <professor>
      <id>65805</id>
      <name>professorName65805</name>
      <street>xxxx_streetname</street>
      <city>city_name</city>
      <state>Minnesota</state>
      <zipcode>92344 </zipcode>
      <birthdate>1983-10-30</birthdate>
      <picture>picture</picture>
      <latitude>1337</latitude>
      <longitude>60</longitude>
      <dept>220</dept>
      <datehired>1943-04-19</datehired>
      <status>9</status>
      <kidnames>boy99</kidnames>
      <kidnames>girl1057</kidnames>
      <kidnames>girl877</kidnames>
      <kidnames>girl99</kidnames>
<aysalary>127000</aysalary>
<monthsummer>3</monthsummer>
</professor>
  <professor>
      <id>47254</id>
      <name>professorName47254</name>
      <street>xxxx_streetname</street>
      <city>city_name</city>
      <state>Pennsylvania</state>
      <zipcode>34306 </zipcode>
      <birthdate>1981-12-03</birthdate>
      <picture>picture</picture>
      <latitude>1904</latitude>
      <longitude>1565</longitude>
      <dept>25</dept>
      <datehired>1982-01-09</datehired>
      <status>0</status>
```

```
    <aysalary>127000</aysalary>
    <monthsummer>1</monthsummer>
  </professor>
  <professor>
    <id>54453</id>
    <name>professorName54453</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Virginia</state>
    <zipcode>23403 </zipcode>
    <birthdate>1971-04-04</birthdate>
    <picture>picture</picture>
    <latitude>1111</latitude>
    <longitude>1753</longitude>
    <dept>10</dept>
    <datehired>1970-05-14</datehired>
    <status>9</status>
    <aysalary>111000</aysalary>
    <monthsummer>3</monthsummer>
  </professor>
  <staff>
    <id>1</id>
    <name>staffName1</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Oklahoma</state>
    <zipcode>41421 </zipcode>
    <birthdate>1989-11-13</birthdate>
    <picture>picture</picture>
    <latitude>362</latitude>
    <longitude>27</longitude>
    <dept>1</dept>
    <datehired>1955-11-28</datehired>
    <status>3</status>
    <kidnames>boy90</kidnames>
    <kidnames>girl90</kidnames>
    <annualsalary>83000</annualsalary>
  </staff>
  <staff>
    <id>2</id>
    <name>staffName2</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Oregon</state>
    <zipcode>56429 </zipcode>
    <birthdate>1957-12-09</birthdate>
    <picture>picture</picture>
    <latitude>1782</latitude>
    <longitude>1530</longitude>
    <dept>1</dept>
    <datehired>1966-11-01</datehired>
```

```
    <status>2</status>
    <kidnames>boy62</kidnames>
    <kidnames>girl62</kidnames>
    <annualsalary>33000</annualsalary>
  </staff>
  <staff>
    <id>3</id>
    <name>staffName3</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Florida</state>
    <zipcode>18456 </zipcode>
    <birthdate>1983-06-28</birthdate>
    <picture>picture</picture>
    <latitude>1011</latitude>
    <longitude>42</longitude>
    <dept>10</dept>
    <datehired>1942-08-30</datehired>
    <status>2</status>
    <kidnames>boy90</kidnames>
    <kidnames>girl29</kidnames>
    <annualsalary>47000</annualsalary>
  </staff>
  <student>
    <id>65990</id>
    <name>studentName65990</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Vermont</state>
    <zipcode>24609 </zipcode>
    <birthdate>1941-04-16</birthdate>
    <picture>picture</picture>
    <latitude>579</latitude>
    <longitude>534</longitude>
    <studentno>454356786</studentno>
    <majordept>1</majordept>
    <advisor>54453</advisor>
  </student>
  <student>
    <id>75001</id>
    <name>studentName75001</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Georgia</state>
    <zipcode>47880 </zipcode>
    <birthdate>1985-07-21</birthdate>
    <picture>picture</picture>
    <latitude>1464</latitude>
    <longitude>179</longitude>
    <studentno>503582122</studentno>
    <majordept>1</majordept>
```

```
    <advisor>59247</advisor>
  </student>
  <student>
    <id>75051</id>
    <name>studentName75051</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Florida</state>
    <zipcode>63335 </zipcode>
    <birthdate>1956-12-18</birthdate>
    <picture>picture</picture>
    <latitude>71</latitude>
    <longitude>755</longitude>
    <studentno>112339778</studentno>
    <majordept>10</majordept>
    <advisor>59247</advisor>
  </student>
  <student>
    <id>117525</id>
    <name>studentName117525</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Oklahoma</state>
    <zipcode>97695 </zipcode>
    <birthdate>1952-08-27</birthdate>
    <picture>picture</picture>
    <latitude>1619</latitude>
    <longitude>900</longitude>
    <studentno>374361804</studentno>
    <majordept>10</majordept>
    <advisor>47254</advisor>
  </student>
  <student>
    <id>108981</id>
    <name>studentName108981</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>West_Virginia</state>
    <zipcode>83589 </zipcode>
    <birthdate>1965-09-02</birthdate>
    <picture>picture</picture>
    <latitude>1867</latitude>
    <longitude>985</longitude>
    <studentno>75613281</studentno>
    <majordept>220</majordept>
    <advisor>65805</advisor>
  </student>
  <student>
    <id>47397</id>
    <name>studentName47397</name>
    <street>xxxx_streetname</street>
```

```
    <city>city_name</city>
    <state>Iowa</state>
    <zipcode>64177 </zipcode>
    <birthdate>1970-10-12</birthdate>
    <picture>picture</picture>
    <latitude>1043</latitude>
    <longitude>486</longitude>
    <studentno>651921317</studentno>
    <majordept>25</majordept>
    <advisor>47254</advisor>
  </student>
  <student>
    <id>71661</id>
    <name>studentName71661</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Idaho</state>
    <zipcode>64177 </zipcode>
    <birthdate>1952-04-17</birthdate>
    <picture>picture</picture>
    <latitude>377</latitude>
    <longitude>859</longitude>
    <studentno>974503561</studentno>
    <majordept>25</majordept>
    <advisor>65805</advisor>
  </student>
  <ta>
    <id>65990</id>
    <name>studentName65990</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Vermont</state>
    <zipcode>24609 </zipcode>
    <birthdate>1941-04-16</birthdate>
    <picture>picture</picture>
    <latitude>579</latitude>
    <longitude>534</longitude>
    <dept>220</dept>
    <datehired>1987-09-29</datehired>
    <status>6</status>
    <semestersalary>12000</semestersalary>
    <apptfraction>.55</apptfraction>
  </ta>
  <ta>
    <id>47397</id>
    <name>studentName47397</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Iowa</state>
    <zipcode>64177 </zipcode>
    <birthdate>1970-10-12</birthdate>
```

```
    <picture>picture</picture>
    <latitude>1043</latitude>
    <longitude>486</longitude>
    <dept>1</dept>
    <datehired>1963-11-27</datehired>
    <status>2</status>
    <semestersalary>16000</semestersalary>
    <apptfraction>.45</apptfraction>
  </ta>
  <ta>
    <id>71661</id>
    <name>studentName71661</name>
    <street>xxxx_streetname</street>
    <city>city_name</city>
    <state>Idaho</state>
    <zipcode>64177 </zipcode>
    <birthdate>1952-04-17</birthdate>
    <picture>picture</picture>
    <latitude>377</latitude>
    <longitude>859</longitude>
    <dept>25</dept>
    <datehired>1963-11-27</datehired>
    <status>9</status>
    <semestersalary>17000</semestersalary>
    <apptfraction>.65</apptfraction>
  </ta>
</XMLSchema>
```

Appendix D

School Database Schema Translation

This Appendix contains the OODB and ORDB schemas of the School RDB translated by MIGROX and Urban and Dietrich [2003].

D.1 ODMG 3.0 ODL of School database mapped by Urban and Dietrich [2003]

```
class Person( extent people key pID) {
attribute string pID; attribute date dob;
attribute string firstName; attribute string lastName;
.  .  .  }
class Student extends Person(extent students){
attribute string status; attribute Department major;
relationship set<CampusClub> memberOf inverse CampusClub::members;
.  .  .  }
class Faculty extends Person(extent facultyMembers){
attribute string rank; attribute Department dept;
relationship set<CampusClub> advisorOf inverse CampusClub::advisor;
Department getChairOf();
.  .  .  }
class CampusClub(extent campusClubs key cID){
attribute string cID; attribute string name; attribute string location;
attribute string phone;
relationship set<Student> members inverse Student::memberOf;
relationship Faculty advisor inverse Faculty::advisorOf;
.  .  .  }
class Department(extent departments key code) {
attribute string code; attribute string name; attribute Faculty deptChair;
attribute set<Student> students; attribute set<Faculty> deptFaculty;
.  .  .  }
```

D.2 ODMG 3.0 ODL of School database generated by MIGROX

```
class campusclub (extent campusclubs key cid) {
attribute string cid; attribute string name; attribute string phone;
attribute string location;
relationship set<student> members inverse student::memberof;
relationship faculty advisor inverse faculty::advisorof};

class department (extent departments key code) {
attribute string code; attribute string name;
relationship set<faculty> deptfuclty inverse faculty::dept;
relationship set<student> students inverse student::major;
relationship faculty deptchair inverse faculty::chairof};

class faculty extends person (extent facultys) {
attribute string rank;
relationship set<campusclub> advisorof inverse campusclub::advisor;
relationship department chairof inverse department::deptchair;
relationship department dept inverse department::deptfuclty};

class person (extent persons key pid) {
attribute string pid; attribute date dob; attribute string firstname;
attribute string lastname;};

class student extends person (extent students) {
attribute string status;
relationship set<campusclub> memberof inverse campusclub::members;
relationship department major inverse department::students};
```

D.3 SQL3 DDL of School database mapped by Urban and Dietrich [2003]

```
create type person_udt as (
pid varchar(11), dob date, firstname varchar(20), lastname varchar(20)) not final
ref is system generated;

create table person of person_udt (
constraint person_pk primary key(pid), ref is oid system generated);

create type faculty_udt under person_udt as (
rank varchar(20),
advisorof ref(campusclub_udt) scope campusclub array[20],
worksin ref(department_udt) scope department,
chairof ref(department_udt) scope department)
not final
method getclubsadvised() returns varchar(25) array[20];
```

```
create table faculty of faculty_udt under person;

create type student_udt under person_udt as (
status varchar(20),
clubs ref(campusclub_udt) scope campusclub array[20],
major ref(department_udt) scope department)not final
method getclubs() returns varchar(25) array[20];
create table student of student_udt under person;

create type campusclub_udt as (
cid varchar(11), name varchar(25), location varchar(25), phone varchar(25),
advisor ref( faculty_udt) scope faculty,
members ref(student_udt) scope student array[100])not final
ref is system generated;
create table campusclub of campusclub_udt (
constraint campusclub_pk primary key(cid),
ref is oid system generated);

create type department_udt as (
code varchar(3), name varchar(40),
deptchair ref(faculty_udt) scope faculty)not final
ref is system generated
method getstudents() returns varchar(40) array[1000],
method getfaculty() returns varchar(40) array[50];
create table department of department_udt (
constraint department_pk primary key(code),
ref is oid system generated);
```

D.4 Oracle 11_g of School database generated by MIGROX

```
create type campusclub_t
/
create type department_t
/
create type faculty_t
/
create type person_t
/
create type student_t
/
create or replace type student_ntt as table of ref student_t;
/
create or replace type faculty_ntt as table of ref faculty_t;
/
create or replace type campusclub_ntt as table of ref campusclub_t;
/
create or replace type campusclub_t as object (
```

```
cid char(10), name char(20), phone char(10), location char(30),
members student_ntt, advisor ref faculty_t) not final;
/
create or replace type department_t as object (
code char(3), name char(20), deptfuclty faculty_ntt, students student_ntt,
deptchair ref faculty_t) not final;
/
create or replace type person_t as object (
pid char(10), dob date, firstname char(10), lastname char(10)) not final;
/
create or replace type faculty_t under person_t (
rank char(5), advisorof campusclub_ntt, chairof ref department_t,
dept ref department_t) final;
/
create or replace type student_t under person_t (
status char(10), memberof campusclub_ntt, major ref department_t) final;
/
create table campusclub of campusclub_t
nested table members store as members_nt
;
create table department of department_t
nested table deptfuclty store as deptfuclty_nt
nested table students store as students_nt
;
create table faculty of faculty_t
nested table advisorof store as advisorof_nt
;
create table student of student_t
nested table memberof store as memberof_nt
;
alter table campusclub add constraint campusclub_pk primary key (cid);
alter table department add constraint department_pk primary key (code);
```

Appendix E

Company Database Schema Translation

This Appendix contains the XML Schema documents translated by MIGROX and Elmasri and Navathe [2006] algorithm, respectively. The documents are mapped from the Company RDB Elmasri and Navathe [2006].

E.1 XML Schema document generated by MIGROX

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:annotation>
<xs:documentation xml:lang ="en"> Generated XML Schema </xs:documentation>
</xs:annotation>
<xs:element name= "XMLSchema">
<xs:complexType>
<xs:sequence>
     <xs:element name = "department" type= "department_t" maxOccurs = "unbounded"/>
     <xs:element name = "employee" type= "employee_t" maxOccurs = "unbounded"/>
     <xs:element name = "project" type= "project_t" maxOccurs = "unbounded"/>
     <xs:element name = "works_on" type= "works_on_t" maxOccurs = "unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:key name= "departmentdnumberPK">
   <xs:selector xpath= ".//department"/>
   <xs:field xpath= "dnumber"/>
</xs:key>
<xs:key name= "employeessnPK">
   <xs:selector xpath= ".//employee"/>
   <xs:field xpath= "ssn"/>
</xs:key>
<xs:key name= "projectpnumberPK">
   <xs:selector xpath= ".//project"/>
```

```
   <xs:field xpath= "pnumber"/>
</xs:key>
<xs:key name= "works_onessnpnoPK">
   <xs:selector xpath= ".//works_on"/>
   <xs:field xpath= "essn"/>
<xs:field xpath= "pno"/>
</xs:key>
<xs:keyref name= "departmentmgrssnFK" refer= "employeessnPK">
   <xs:selector xpath= ".//department"/>
   <xs:field xpath= "mgrssn"/>
</xs:keyref>
<xs:keyref name= "employeednoFK" refer= "departmentdnumberPK">
   <xs:selector xpath= ".//employee"/>
   <xs:field xpath= "dno"/>
</xs:keyref>
<xs:keyref name= "employeesuperssnFK" refer= "employeessnPK">
   <xs:selector xpath= ".//employee"/>
   <xs:field xpath= "superssn"/>
</xs:keyref>
<xs:keyref name= "projectdnumFK" refer= "departmentdnumberPK">
   <xs:selector xpath= ".//project"/>
   <xs:field xpath= "dnum"/>
</xs:keyref>
<xs:keyref name= "works_onessnFK" refer= "employeessnPK">
   <xs:selector xpath= ".//works_on"/>
   <xs:field xpath= "essn"/>
</xs:keyref>
<xs:keyref name= "works_onpnoFK" refer= "projectpnumberPK">
<xs:selector xpath= ".//works_on"/>
<xs:field xpath= "pno"/>
</xs:keyref>
</xs:element>
<xs:complexType name = "department_t">
   <xs:sequence>
     <xs:element name = "dname" type= "xs:string"/>
     <xs:element name = "dnumber" type= "xs:int"/>
     <xs:element name = "mgrssn" type= "xs:int"/>
     <xs:element name = "mgrstartdate" type= "xs:date" minOccurs= "0"/>
     <xs:element name = "locations" type= "xs:string" maxOccurs= "unbounded"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "dependent_t">
   <xs:sequence>
     <xs:element name = "dependent_name" type= "xs:string"/>
     <xs:element name = "sex" type= "xs:string"/>
     <xs:element name = "bdate" type= "xs:date" minOccurs= "0"/>
     <xs:element name = "relationship" type= "xs:string" minOccurs= "0"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "employee_t">
   <xs:sequence>
```

```
     <xs:element name = "fname" type= "xs:string"/>
     <xs:element name = "minit" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "lname" type= "xs:string"/>
     <xs:element name = "ssn" type= "xs:int"/>
     <xs:element name = "bdate" type= "xs:date" minOccurs= "0"/>
     <xs:element name = "address" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "sex" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "salary" type= "xs:int" minOccurs= "0"/>
     <xs:element name = "superssn" type= "xs:int" minOccurs= "0"/>
     <xs:element name = "dno" type= "xs:int"/>
     <xs:element name = "hasDependent" type= "dependent_t" minOccurs= "0"
maxOccurs= "unbounded"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "project_t">
   <xs:sequence>
     <xs:element name = "pname" type= "xs:string"/>
     <xs:element name = "pnumber" type= "xs:int"/>
     <xs:element name = "plocation" type= "xs:string" minOccurs= "0"/>
     <xs:element name = "dnum" type= "xs:int"/>
   </xs:sequence>
</xs:complexType>
<xs:complexType name = "works_on_t">
   <xs:sequence>
     <xs:element name = "essn" type= "xs:int"/>
     <xs:element name = "pno" type= "xs:int"/>
     <xs:element name = "hours" type= "xs:int"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

E.2 XML Schema document mapped by Elmasri and Navathe [2006]

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xsd:annotation>
<xsd:documentation xml:lang ="en"> Company Schema </xsd:documentation>
</xsd:annotation>
<xsd:element name= "company">
<xsd:complexType>
<xsd:sequence>
     <xsd:element name = "department" type= "Department" minOccurs = "0"
maxOccurs = "unbounded"/>
     <xsd:element name = "employee" type= "Employee" minOccurs = "0"
maxOccurs = "unbounded">
       <xsd:unique name= "dependentNameUnique">
```

```
          <xsd:selector xpath= "employeeDependent"/>
          <xsd:field xpath= "employeeName"/>
       </xsd:unique">
     </xsd:element">
     <xsd:element name = "project" type= "Project" minOccurs = "0"
maxOccurs = "unbounded"/>
</xsd:sequence>
</xsd:complexType>
<xsd:unique name= "departmentNameUnique">
   <xsd:selector xpath= "department"/>
   <xsd:field xpath= "departmentName"/>
</xsd:unique">
<xsd:unique name= "projectNameUnique">
   <xsd:selector xpath= "project"/>
   <xsd:field xpath= "projectName"/>
</xsd:unique">
<xsd:key name= "projectNumberKey">
   <xsd:selector xpath= "project"/>
   <xsd:field xpath= "projectNumber"/>
</xsd:key>
<xsd:key name= "departmentNumberKey">
   <xsd:selector xpath= "department"/>
   <xsd:field xpath= "departmentNumber"/>
</xsd:key>
<xsd:key name= "employeeSSNKey">
   <xsd:selector xpath= "employee"/>
   <xsd:field xpath= "employeeSSN"/>
</xsd:key>
<xsd:keyref name= "departmentManagerSSNKeyRef" refer= "employeeSSNKey">
   <xsd:selector xpath= "department"/>
   <xsd:field xpath= "departmentManagerSSN"/>
</xsd:keyref>
<xsd:keyref name= "employeeDepartmentNumberKeyRef" refer= "departmentNumberKey">
   <xsd:selector xpath= "employee"/>
   <xsd:field xpath= "employeeDepartmentNumber"/>
</xsd:keyref>
<xsd:keyref name= "employeeSupervisorSSNKeyRef" refer= "employeeSSNKey">
   <xsd:selector xpath= "employee"/>
   <xsd:field xpath= "employeeSupervisorSSN"/>
</xsd:keyref>
<xsd:keyref name= "projectDepartmentNumberKeyRef" refer= "departmentNumberKey">
   <xsd:selector xpath= "project"/>
   <xsd:field xpath= "projectDepartmentNumber"/>
</xsd:keyref>
<xsd:keyref name= "projectWorkerSSNKeyRef" refer= "employeeSSNKey">
   <xsd:selector xpath= "project/projectWorker"/>
   <xsd:field xpath= "SSN"/>
</xsd:keyref>
<xsd:keyref name= "employeeWorksOnProjectNumberKeyRef" refer= "projectNumberKey">
   <xsd:selector xpath= "employee/employeeWorksOn"/>
   <xsd:field xpath= "projectNumber"/>
```

```
</xsd:keyref>
</xsd:element>
<xsd:complexType name = "Department">
   <xsd:sequence>
     <xsd:element name = "departmentName" type= "xsd:string"/>
     <xsd:element name = "departmentNumber" type= "xsd:string"/>
     <xsd:element name = "departmentManagerSSN" type= "xsd:string"/>
     <xsd:element name = "departmentManagerStartDate" type= "xsd:date"/>
     <xsd:element name = "departmentLocations" type= "xsd:string" minOccurs= "0"
maxOccurs= "unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "Employee">
   <xsd:sequence>
     <xsd:element name = "employeeName" type= "Name"/>
     <xsd:element name = "employeeSsn" type= "xsd:string"/>
     <xsd:element name = "employeeSex" type= "xsd:string"/>
     <xsd:element name = "employeeSalary" type= "xsd:unsignedInt"/>
     <xsd:element name = "employeeBirthdate" type= "xsd:date"/>
     <xsd:element name = "employeeDepartmentNumber" type= "xsd:string"/>
     <xsd:element name = "employeeSuperSSN" type= "xsd:string/>
     <xsd:element name = "employeeAddress" type= "Address"/>
     <xsd:element name = "employeeWorksOn" type= "WorksOn" minOccurs= "1"
maxOccurs= "unbounded"/>
     <xsd:element name = "employeeDependent" type= "Dependent" minOccurs= "0"
maxOccurs= "unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "Project">
   <xsd:sequence>
     <xsd:element name = "projectName" type= "xsd:string"/>
     <xsd:element name = "projectNumber" type= "xsd:string"/>
     <xsd:element name = "projectLocation" type= "xsd:string"/>
     <xsd:element name = "projectDepartmentNumber" type= "xsd:string"/>
     <xsd:element name = "projectWorker" type= "Worker" minOccurs= "1"
maxOccurs= "unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "Dependent">
   <xsd:sequence>
     <xsd:element name = "dependentName" type= "xsd:string"/>
     <xsd:element name = "dependentSsex" type= "xsd:string"/>
     <xsd:element name = "dependentBirthDate" type= "xsd:date"/>
     <xsd:element name = "dependentRelationship" type= "xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "Address">
   <xsd:sequence>
     <xsd:element name = "name" type= "xsd:string"/>
     <xsd:element name = "street" type= "xsd:string"/>
     <xsd:element name = "city" type= "xsd:string"/>
```

```
      <xsd:element name = "state" type= "xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "Name">
    <xsd:sequence>
      <xsd:element name = "firstName" type= "xsd:string"/>
      <xsd:element name = "middleName" type= "xsd:string"/>
      <xsd:element name = "lastName" type= "xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "Worker">
    <xsd:sequence>
      <xsd:element name = "SSN" type= "xsd:string"/>
      <xsd:element name = "hours" type= "xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name = "WorksOn">
    <xsd:sequence>
      <xsd:element name = "projectName" type= "xsd:string"/>
      <xsd:element name = "hours" type= "xsd:float"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Appendix F

Query Plans of RDB UniDB

This Appendix includes the query plan tables generated from Oracle for executing the RDB UniDB queries used in the experiment of performance comparison.

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1	44	1 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	44	1 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	DEPT_PK	1		0 (0)	00:00:01
Predicate Information (identified by operation id): 2 - access("DEPTNO"=1)						

Table F.1: Plan table for relational SINGLE-EXACT query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1	102	3 (0)	00:00:01
1	NESTED LOOPS		1	102	3 (0)	00:00:01
2	TABLE ACCESS BY INDEX ROWID	STAFF	1	26	2 (0)	00:00:01
* 3	INDEX UNIQUE SCAN	STAFF_PK	1		1 (0)	00:00:01
4	TABLE ACCESS BY INDEX ROWID	PERSON	1	76	1 (0)	00:00:01
* 5	INDEX UNIQUE SCAN	PERSON_PK	1		0 (0)	00:00:01
Predicate Information (identified by operation id): 3 - access("S"."ID"=2) 5 - access("P"."ID"=2)						

Table F.2: Plan table for relational HIER-EXACT query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		2014	78546	18 (6)	00:00:01
* 1	TABLE ACCESS FULL	PROFESSOR	2014	78546	18 (6)	00:00:01
Predicate Information (identified by operation id): 1 - filter("P"."AYSALARY"*(9+"P"."MONTHSUMMER")/9.0>=145000) Note–dynamic sampling used for this statement						

Table F.3: Plan table for relational SINGLE-METH query

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		2790	280K		588 (100)	00:00:08
1	SORT UNIQUE		2790	280K	664K	588 (100)	00:00:08
2	UNION-ALL						
3	NESTED LOOPS						
4	NESTED LOOPS		1	90		2 (0)	00:00:01
5	TABLE ACCESS BY INDEX ROWID	STAFF	1	26		1 (0)	00:00:01
* 6	INDEX RANGE SCAN	STAFF_ANNUALSALARY	1			1 (0)	00:00:01
* 7	INDEX UNIQUE SCAN	PERSON_PK	1			0 (0)	00:00:01
8	TABLE ACCESS BY INDEX ROWID	PERSON	1	64		1 (0)	00:00:01
* 9	HASH JOIN		2788	280K		497 (1)	00:00:06
* 10	TABLE ACCESS FULL	PROFESSOR	2788	106K		18 (6)	00:00:01
11	TABLE ACCESS FULL	PERSON	114K	7180K		479 (1)	00:00:06
12	NESTED LOOPS						
13	NESTED LOOPS		1	103		18 (0)	00:00:01
* 14	TABLE ACCESS FULL	TA	1	39		17 (0)	00:00:01
* 15	INDEX UNIQUE SCAN	PERSON_PK	1			0 (0)	00:00:01
16	TABLE ACCESS BY INDEX ROWID	PERSON	1	64		1 (0)	00:00:01

Predicate Information (identified by operation id):
6 - access("S"."ANNUALSALARY">=140000)
7 - access("P"."ID"="S"."ID")
9 - access("P"."ID"="F"."ID")
10 - filter("F"."AYSALARY"*(9+"F"."MONTHSUMMER")/9.0>=140000)
14 - filter("APPTFRACTION"*(2*"T"."SEMESTERSALARY")>=140000)
15 - access("P"."ID"="T"."ID")
Note–dynamic sampling used for this statement

Table F.4: Plan table for relational HIER-METH query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		313	27544	7 (15)	00:00:01
* 1	HASH JOIN		313	27544	7 (15)	00:00:01
2	TABLE ACCESS FULL	DEPARTMENT	250	11000	3 (0)	00:00:01
3	TABLE ACCESS FULL	DEPARTMENT	250	11000	3 (0)	00:00:01

Predicate Information (identified by operation id):
1 - access("D1"."BUDGET"="D2"."BUDGET")
filter("D1"."DEPTNO" < "D2"."DEPTNO")
Note—dynamic sampling used for this statement

Table F.5: Plan table for relational SINGLE-JOIN query

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		1	142		2087 (1)	00:00:26
1	NESTED LOOPS		1	142		2087 (1)	00:00:26
2	NESTED LOOPS		1	129		2087 (1)	00:00:26
* 3	HASH JOIN		1	116	1336K	2087 (1)	00:00:26
* 4	HASH JOIN		12834	1178K	1056K	1778 (1)	00:00:22
* 5	HASH JOIN		12834	902K	5392K	1483 (1)	00:00:18
6	TABLE ACCESS FULL	PERSON	114K	4039K		479 (1)	00:00:06
7	TABLE ACCESS FULL	PERSON	114K	4039K		479 (1)	00:00:06
8	TABLE ACCESS FULL	EMPLOYEE	86498	1858K		103 (1)	00:00:02
9	TABLE ACCESS FULL	EMPLOYEE	86498	1858K		103 (1)	00:00:02
* 10	INDEX UNIQUE SCAN	TA_PK	1	13		0 (0)	00:00:01
* 11	INDEX UNIQUE SCAN	TA_PK	1	13		0 (0)	00:00:01

Predicate Information (identified by operation id):
3 - access("E1"."DATEHIRED"="E2"."DATEHIRED" AND "P2"."ID"="E2"."ID")
4 - access("P1"."ID"="E1"."ID")
5 - access("P1"."ZIPCODE"="P2"."ZIPCODE")
filter("P1"."ID" < "P2"."ID")
10 - access("P1"."ID"="T1"."ID")
11 - access("P2"."ID"="T2"."ID")
Note–dynamic sampling used for this statement

Table F.6: Plan table for relational HIER-JOIN query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		497	53676	489 (3)	00:00:06
1	NESTED LOOPS		497	53676	489 (3)	00:00:06
2	NESTED LOOPS		497	47215	489 (3)	00:00:06
3	TABLE ACCESS FULL	PERSON	114K	8527K	479 (1)	00:00:06
* 4	INDEX UNIQUE SCAN	KIDS_PK	1	19	0 (0)	00:00:01
* 5	INDEX UNIQUE SCAN	STAFF_PK	1	13	0 (0)	00:00:01

Predicate Information (identified by operation id):
4 - access("P"."ID"="K"."ID" AND "K"."KIDNAME"='boy90')
5 - access("S"."ID"="K"."ID")
Note–dynamic sampling used for this statement

Table F.7: Plan table for relational SET-ELEMENT query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		10	1270	27 (8)	00:00:01
1	NESTED LOOPS					
2	NESTED LOOPS		10	1270	27 (8)	00:00:01
3	NESTED LOOPS		10	510	17 (12)	00:00:01
4	NESTED LOOPS		497	15904	17 (12)	00:00:01
5	TABLE ACCESS FULL	STAFF	25000	317K	15 (0)	00:00:01
* 6	INDEX UNIQUE SCAN	KIDS_PK	1	19	0 (0)	00:00:01
* 7	INDEX UNIQUE SCAN	KIDS_PK	1	19	0 (0)	00:00:01
* 8	INDEX UNIQUE SCAN	PERSON_PK	1		0 (0)	00:00:01
9	TABLE ACCESS BY INDEX ROWID	PERSON	1	76	1 (0)	00:00:01

Predicate Information (identified by operation id):
6 - access("E"."ID"="K1"."ID" AND "K1"."KIDNAME"='girl90')
7 - access("E"."ID"="K2"."ID" AND "K2"."KIDNAME"='boy90')
8 - access("E"."ID"="P"."ID")
Note–dynamic sampling used for this statement

Table F.8: Plan table for relational SET-AND query

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		79638	6532K		998 (1)	00:00:12
* 1	HASH JOIN		79638	6532K		998 (1)	00:00:12
2	TABLE ACCESS FULL	DEPARTMENT	250	5250		3 (0)	00:00:01
* 3	HASH JOIN		79638	4899K	2960K	994 (1)	00:00:12
4	TABLE ACCESS FULL	STUDENT	79638	2022K		103 (1)	00:00:02
5	TABLE ACCESS FULL	PERSON	114K	4151K		479 (1)	00:00:06

Predicate Information (identified by operation id):
1 - access("S"."MAJORDEPT"="D"."DEPTNO")
3 - access("P"."ID"="S"."ID")
Note–dynamic sampling used for this statement

Table F.9: Plan table for relational 1HOP-NONE query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1	82	6 (0)	00:00:01
1	NESTED LOOPS					
2	NESTED LOOPS		1	82	6 (0)	00:00:01
3	NESTED LOOPS		1	51	5 (0)	00:00:01
4	TABLE ACCESS BY INDEX ROWID	PERSON	1	25	4 (0)	00:00:01
* 5	INDEX RANGE SCAN	PERSON_NAME	1		3 (0)	00:00:01
6	TABLE ACCESS BY INDEX ROWID	STUDENT	1	26	1 (0)	00:00:01
* 7	INDEX UNIQUE SCAN	STUD_PK	1		0 (0)	00:00:01
* 8	INDEX UNIQUE SCAN	DEPT_PK	1		0 (0)	00:00:01
9	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	31	1 (0)	00:00:01

Predicate Information (identified by operation id):
5 - access("P"."NAME"='studentName75001')
7 - access("P"."ID"="S"."ID")
8 - access("S"."MAJORDEPT"="D"."DEPTNO")
Note–dynamic sampling used for this statement

Table F.10: Plan table for relational 1HOP-ONE query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		285	20520	391 (1)	00:00:05
1	NESTED LOOPS					
2	NESTED LOOPS		285	20520	391 (1)	00:00:05
* 3	HASH JOIN		285	13395	106 (2)	00:00:02
4	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	21	2 (0)	00:00:01
* 5	INDEX RANGE SCAN	DEPARTMENT_NAME	1		1 (0)	00:00:01
6	TABLE ACCESS FULL	STUDENT	71169	1807K	103 (1)	00:00:02
* 7	INDEX UNIQUE SCAN	PERSON_PK	1		0 (0)	00:00:01
8	TABLE ACCESS BY INDEX ROWID	PERSON	1	25	1 (0)	00:00:01

Predicate Information (identified by operation id):
3 - access("S"."MAJORDEPT"="D"."DEPTNO")
5 - access("D"."NAME"='deptname1')
7 - access("P"."ID"="S"."ID")
Note–dynamic sampling used for this statement

Table F.11: Plan table for relational 1HOP-MANY query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		2	224	105 (1)	00:00:02
1	NESTED LOOPS					
2	NESTED LOOPS		2	224	105 (1)	00:00:02
3	NESTED LOOPS		2	182	103 (1)	00:00:02
* 4	TABLE ACCESS FULL	COURSESECTION	530	34450	103 (1)	00:00:02
* 5	INDEX UNIQUE SCAN	COURSE_PK	1	26	0 (0)	00:00:01
* 6	INDEX UNIQUE SCAN	DEPT_PK	1		0 (0)	00:00:01
7	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	21	1 (0)	00:00:01
Predicate Information (identified by operation id): 4 - filter("X"."ROOMNO"=50) 5 - access("X"."DEPTNO"="C"."DEPTNO" AND "X"."COURSENO"="C"."COURSENO") 6 - access("C"."DEPTNO"="D"."DEPTNO") Note–dynamic sampling used for this statement						

Table F.12: Plan table for relational 1HOP-MANY query

Appendix G

Query Plans of ORDB UniDB

This Appendix includes the query plan tables generated from Oracle for executing the ORDB UniDB queries used in the experiment of performance comparison.

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1	44	1 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	44	1 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	DEPARTMENT_PK	1		0 (0)	00:00:01
Predicate Information (identified by operation id): 2 - access("DEPTNO"=1)						

Table G.1: Plan table for ORDB SINGLE-EXACT query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1	38	2 (0)	00:00:01
1	TABLE ACCESS BY INDEX ROWID	STAFF	1	38	2 (0)	00:00:01
* 2	INDEX UNIQUE SCAN	STAFF_PK	1		1 (0)	00:00:01
Predicate Information (identified by operation id): 2 - access("ID"=2)						

Table G.2: Plan table for ORDB HIER-EXACT query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		2014	78546	18 (6)	00:00:01
* 1	TABLE ACCESS FULL	PROFESSOR	2014	78546	18 (6)	00:00:01
Predicate Information (identified by operation id): 1 - filter("P"."AYSALARY"*(9+"P"."MONTHSUMMER")/9.0>=145000) Note–dynamic sampling used for this statement						

Table G.3: Plan table for ORDB SINGLE-METH query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		2208	86112	205 (1)	00:00:03
* 1	TABLE ACCESS FULL	PROFESSOR	2208	86112	205 (1)	00:00:03
Predicate Information (identified by operation id): 1 - filter("P"."AYSALARY"*(9+"P"."MONTHSUMMER")/9.0>=145000) Note–dynamic sampling used for this statement						

Table G.4: Plan table for ORDB SINGLE-METH query

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		75000	11M		3416 (70)	00:00:41
1	SORT UNIQUE		75000	11M	28M	3416 (70)	00:00:41
2	UNION-ALL						
* 3	FILTER						
4	TABLE ACCESS FULL	STAFF	25000	3344K		208 (2)	00:00:03
* 5	TABLE ACCESS FULL	KIDNAMES_STAFF_NT	2	46		69 (2)	00:00:01
* 6	FILTER						
7	TABLE ACCESS FULL	PROFESSOR	25000	4345K		242 (2)	00:00:03
* 8	TABLE ACCESS FULL	KIDNAMES_PROFESSOR_NT	2	46		69 (2)	00:00:01
* 9	TABLE ACCESS FULL	TEACHES_PROFESSOR_NT	3	51		68 (0)	00:00:01
* 10	TABLE ACCESS FULL	ADVISES_NT	3	51		171 (1)	00:00:03
* 11	FILTER						
12	TABLE ACCESS FULL	TA	25000	3881K		208 (2)	00:00:03
* 13	TABLE ACCESS FULL	KIDNAMES_TA_NT	1	16		2 (0)	00:00:01
* 14	TABLE ACCESS FULL	TEACHES_TA_NT	3	51		68 (0)	00:00:01
Predicate Information (identified by operation id): 3 - filter("STAFF_T"."SALARY"("S"."SYS_NC_ROWINFO$")>=140000) 5 - filter("NESTED_TABLE_ID"=:B1) 6 - filter("PROFESSOR_T"."SALARY"("P"."SYS_NC_ROWINFO$")>=140000) 8 - filter("NESTED_TABLE_ID"=:B1) 9 - filter("NESTED_TABLE_ID"=:B1) 10 - filter("NESTED_TABLE_ID"=:B1) 11 - filter("TA_T"."SALARY"("T"."SYS_NC_ROWINFO$")>=140000) 13 - filter("NESTED_TABLE_ID"=:B1) 14 - filter("NESTED_TABLE_ID"=:B1)							

Table G.5: Plan table for ORDB HIER-METH query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		313	27544	7 (15)	00:00:01
* 1	HASH JOIN		313	27544	7 (15)	00:00:01
2	TABLE ACCESS FULL	DEPARTMENT	250	11000	3 (0)	00:00:01
3	TABLE ACCESS FULL	DEPARTMENT	250	11000	3 (0)	00:00:01
Predicate Information (identified by operation id): 1 - access("D1"."BUDGET"="D2"."BUDGET") filter("D1"."DEPTNO" < "D2"."DEPTNO") Note–dynamic sampling used for this statement						

Table G.6: Plan table for ORDB SINGLE-JOIN query

Id	Operation	Name	Rows	Bytes	TempSpc	Cost(%CPU)	Time
0	SELECT STATEMENT		1	90		556 (1)	00:00:07
* 1	HASH JOIN		1	90	1504K	556 (1)	00:00:07
2	TABLE ACCESS FULL	TA	26881	1181K		205 (1)	00:00:03
3	TABLE ACCESS FULL	TA	26881	1181K		205 (1)	00:00:03
Predicate Information (identified by operation id): 1 - access("T1"."DATEHIRED"="T2"."DATEHIRED" AND "T1"."ZIPCODE" = "T2"."ZIPCODE") filter("T1"."ID" < "T2"."ID") Note–dynamic sampling used for this statement							

Table G.7: Plan table for ORDB HIER-JOIN query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		279	28458	240 (1)	00:00:03
* 1	HASH JOIN		279	28458	240 (1)	00:00:03
* 2	TABLE ACCESS FULL	KIDNAMES_STAFF_NT	279	4464	69 (2)	00:00:01
3	TABLE ACCESS FULL	STAFF	20802	1747K	171 (1)	00:00:03
Predicate Information (identified by operation id): 1 - access("K"."NESTED_TABLE_ID"="S"."SYS_NC0001500016$") 2 - filter('K'."KIDNAME"='boy90') Note–dynamic sampling used for this statement						

Table G.8: Plan table for ORDB SET-ELEMENT query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		4	420	309 (1)	00:00:04
* 1	HASH JOIN		4	420	309 (1)	00:00:04
* 2	TABLE ACCESS FULL	KIDNAMES_STAFF_NT	279	4464	69 (2)	00:00:01
* 3	HASH JOIN		279	24831	240 (1)	00:00:03
* 4	TABLE ACCESS FULL	KIDNAMES_STAFF_NT	279	4464	69 (2)	00:00:01
5	TABLE ACCESS FULL	STAFF	20802	1482K	171 (1)	00:00:03

Predicate Information (identified by operation id):
1 - access("K2"."NESTED_TABLE_ID"="S"."SYS_NC00015000016$")
2 - filter("K2"."KIDNAME"='boy90')
3 - access("K1"."NESTED_TABLE_ID"="S"."SYS_NC00015000016$")
4 - filter("K1"."KIDNAME"='girl90')
Note–dynamic sampling used for this statement

Table G.9: Plan table for ORDB SET-AND query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		62917	5837K	551 (1)	00:00:07
* 1	HASH JOIN RIGHT OUTER		62917	5837K	551 (1)	00:00:07
2	TABLE ACCESS FULL	DEPARTMENT	250	7750	3 (0)	00:00:01
3	TABLE ACCESS FULL	STUDENT	62917	3932K	547 (1)	00:00:07

Predicate Information (identified by operation id):
1 - access("P000003$"."SYS_NC_OID$"(+)="S"."MAJOR")
Note–dynamic sampling used for this statement

Table G.10: Plan table for ORDB 1HOP-NONE query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1	93	3 (0)	00:00:01
1	NESTED LOOPS OUTER		1	93	3 (0)	00:00:01
2	TABLE ACCESS BY INDEX ROWID	STUDENT	1	52	2 (0)	00:00:01
* 3	INDEX RANGE SCAN	STUDENT_NAME	1		1 (0)	00:00:01
4	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	41	1 (0)	00:00:01
* 5	INDEX UNIQUE SCAN	SYS_C009892	1		0 (0)	00:00:01

Predicate Information (identified by operation id):
3 - access("NAME"='studentName75001')
5 - access("P000003$"."SYS_NC_OID$"(+)="S"."MAJOR")
Note–dynamic sampling used for this statement

Table G.11: Plan table for ORDB 1HOP-ONE query

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		960	52800	15 (0)	00:00:01
1	NESTED LOOPS					
2	NESTED LOOPS		960	52800	15 (0)	00:00:01
3	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	18	2 (0)	00:00:01
* 4	INDEX RANGE SCAN	DEPARTMENT_NAME	1		1 (0)	00:00:01
* 5	INDEX RANGE SCAN	STUDENTS1_NTAB_IX	960		2 (0)	00:00:01
6	TABLE ACCESS BY INDEX ROWID	STUDENTS_NT1	960	35520	13 (0)	00:00:01

Predicate Information (identified by operation id):
4 - access("D"."NAME"='deptname1')
5 - access("ST"."NESTED_TABLE_ID"="D"."SYS_NC0001400015$")
Note–dynamic sampling used for this statement

Table G.12: Plan table for ORDB 1HOP-MANY query (Variant A)

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		251	17570	279 (0)	00:00:04
1	NESTED LOOPS					
2	NESTED LOOPS		251	17570	279 (0)	00:00:04
3	TABLE ACCESS BY INDEX ROWID	DEPARTMENT	1	18	2 (0)	00:00:01
* 4	INDEX RANGE SCAN	DEPARTMENT_NAME	1		1 (0)	00:00:01
* 5	INDEX RANGE SCAN	STUDENT_MAJOR_I	251		2 (0)	00:00:01
6	TABLE ACCESS BY INDEX ROWID	STUDENT	251	13052	277 (0)	00:00:04

Predicate Information (identified by operation id):
4 - access("NAME"='deptname1')
5 - access("P000003$"."SYS_NC_OID$"="S"."MAJOR")
Note–dynamic sampling used for this statement

Table G.13: Plan table for ORDB 1HOP-MANY query (Variant B)

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		1160K	77M	388K (2)	01:17:46
1	NESTED LOOPS		1160K	77M	388K (2)	01:17:46
* 2	HASH JOIN		14203	943K	34 (3)	00:00:01
3	TABLE ACCESS FULL	DEPARTMENT	250	7750	3 (0)	00:00:01
4	TABLE ACCESS FULL	OFFERS_NT	14203	513K	30 (0)	00:00:01
* 5	COLLECTION ITERATOR PICKLER FETCH					
Predicate Information (identified by operation id): 2 - access("CO"."NESTED_TABLE_ID"="D"."SYS_NC0001000011$") 5 - filter(VALUE(KOKBF$)=50) Note–dynamic sampling used for this statement						

Table G.14: Plan table for ORDB 1HOP-MANY query (Variant A)

Id	Operation	Name	Rows	Bytes	Cost(%CPU)	Time
0	SELECT STATEMENT		530	71020	296 (1)	00:00:04
* 1	HASH JOIN RIGHT OUTER		530	71020	296 (1)	00:00:04
2	TABLE ACCESS FULL	DEPARTMENT	250	7750	3 (0)	00:00:01
* 3	HASH JOIN OUTER		530	54590	293 (1)	00:00:04
4	TABLE ACCESS BY INDEX ROWID	COURSESECTION	530	34980	224 (0)	00:00:03
* 5	INDEX RANGE SCAN	COURSESECTION_ROOMNO	530		2 (0)	00:00:01
6	TABLE ACCESS FULL	COURSE	12452	449K	68 (0)	00:00:01
Predicate Information (identified by operation id): 1 - access("P000005$"."SYS_NC_OID$"(+)="DEPT") 3 - access("P000003$"."SYS_NC_OID$"(+)="S"."COURSE") 5 - access("S"."ROOMNO"=50) Note–dynamic sampling used for this statement						

Table G.15: Plan table for ORDB 1HOP-MANY query (Variant B)

Appendix H

Created Indexes for UniDB

This Appendix contains the indexes that have been created for the RDB and ORDB in the experiment of performance comparison.

H.1 Indexes for RDB

```
create index department_name on department (name);
create index department_building on department (building);
create index department_budget on department (budget);
create index person_name on person (name);
create index person_dob on person (birthdate);
create index person_zipcode on person (zipcode);
create index person_street on person (street);
create index person_city on person (city);
create index person_state on person (state);
create index employee_datehired on employee (datehired);
create index staff_annualsalary on staff (annualsalary);
create index professor_aysalary on professor (aysalary);
create index professor_monthsummer on professor (monthsummer);
create index ta_semestersalary on ta (semestersalary);
create index ta_apptfraction on ta (apptfraction);
create index kids_kidname on kids (kidname);
create index coursesection_nostudentson on coursesection (nostudents);
create index coursesection_roomno on coursesection (roomno);

create index department_chair on department (chair);
create index employee_dept on employee (dept);
create index student_majordept on student (majordept);
create index student_advisor on student (advisor);
create index course_id on course (deptno);
create index coursesection_deptno_courseno on coursesection (deptno, courseno);
create index coursesection_instructorid on coursesection (instructorid);
create index kids_id on kids (id);
create index enrolled_studentid on enrolled (studentid);
create index enrolled_deptno_courseno on enrolled (deptno, courseno,
```

```
sectionno, semester);
```

H.2 Indexes for ORDB

```
create index course_uoid on course (uoid);
create index coursesection_uoid on coursesection (uoid);
create index department_uoid on department (uoid);
create index enrolled_uoid on enrolled (uoid);
create index professor_uoid on professor (uoid);
create index staff_uoid on staff (uoid);
create index student_uoid on student (uoid);
create index ta_uoid on ta (uoid);

alter table department add constraint department_pk primary key (deptno);
alter table professor add constraint professor_pk primary key (id);
alter table staff add constraint staff_pk primary key (id);
alter table student add constraint student_pk primary key (id);
alter table ta add constraint ta_pk primary key (id);

create index department_chair_i on department (chair);
create index enrolled_student_i on enrolled (student);
create index enrolled_section_i on enrolled (section);
create index staff_worksin_i on staff (worksin);
create index ta_worksin_i on ta (worksin);
create index professor_worksin_i on professor (worksin);
create index professor_leads_i on professor (leads);
create index student_major_i on student (major);
create index student_advisor_i on student (advisor);
create index course_dept_i on course (dept);
create index coursesection_course_i on coursesection (course);

create index department_name on department (name);
create index department_building on department (building);
create index department_budget on department (budget);

create index student_name on student (name);
create index student_dob on student (birthdate);
create index student_zipcode on student (zipcode);
create index student_street on student (street);
create index student_city on student (city);
create index student_state on student (state);

create index staff_name on staff (name);
create index staff_dob on staff (birthdate);
create index staff_zipcode on staff (zipcode);
create index staff_street on staff (street);
create index staff_city on staff (city);
create index staff_state on staff (state);
create index staff_annualsalary on staff (annualsalary);
```

```
create index staff_datehired on staff (datehired);
create index staff_kids on kidnames_staff_nt (NESTED_TABLE_ID, kidname);

create index professor_name on professor (name);
create index professor_dob on professor (birthdate);
create index professor_zipcode on professor (zipcode);
create index professor_street on professor (street);
create index professor_city on professor (city);
create index professor_state on professor (state);
create index professor_aysalary on professor (aysalary);
create index professor_monthsummer on professor (monthsummer);
create index professor_datehired on professor (datehired);
create index professor_kids on kidnames_professor_nt (NESTED_TABLE_ID, kidname);

create index ta_name on ta (name);
create index ta_dob on ta (birthdate);
create index ta_zipcode on ta (zipcode);
create index ta_street on ta (street);
create index ta_city on ta (city);
create index ta_state on ta (state);
create index ta_semestersalary on ta (semestersalary);
create index ta_apptfraction on ta (apptfraction);
create index ta_datehired on ta (datehired);
create index ta_kids on kidnames_ta_nt (NESTED_TABLE_ID, kidname);

create index sections_ntab_ix on sections_nt (NESTED_TABLE_ID);
create index students_ntab_ix on students_nt (NESTED_TABLE_ID);
create index offers_ntab_ix on offers_nt (NESTED_TABLE_ID);
create index employees_ntab_ix on employees_nt (NESTED_TABLE_ID);
create index students1_ntab_ix on students_nt1 (NESTED_TABLE_ID);
create index advises_ntab_ix on advises_nt (NESTED_TABLE_ID);
create index teaches_professor_ntab_ix on teaches_professor_nt (NESTED_TABLE_ID);
create index taken_ntab_ix on taken_nt (NESTED_TABLE_ID);
create index teaches_ta_ntab_ix on teaches_ta_nt (NESTED_TABLE_ID);
create index coursesection_nostudentson on coursesection (nostudents);
create index coursesection_roomno on coursesection (roomno);

create index staff_sal on staff p (p.salary());
create index ta_sal on ta p (p.salary());
create index professor_sal on professor p (p.salary());
```

Appendix I

Glossary

A

ADT	Abstract data type

B

BLOB	Binary large object

C

CAC	Composite attribute class
CDM	Canonical data model
CLOB	Character large object
CPU	Central processing unit

D

DBFE	Database forward engineering
DBRE	Database reverse engineering
DBMS	Database management system
DDL	Data definition language
DFK	Disjoint foreign key
DML	Data manipulation language
DTD	Document type definition

E

EER	Extended entity relationship
ER	Entity relationship

J

JDO	Java data objects

M

MAC	Multi-valued attribute class
MIGROX	migrating an RDB into object-based and XML databases

O

ODL	Object definition language
ODMG	Object database management group
OID	Object identifer
OIF	Object interchange format
OO	Object-oriented
OODB	Object-oriented database
OODBMS	Object-oriented DBMS
OOPL	OO programming language
ORDB	Object-relational database
OQL	Object query language
ORDBMS	Object-relational DBMS

R

RCC	Regular component class
RDB	Relational database
RDBMS	Relational database management system
RRC	Regular relationship class
RST	Regular strong class
RSR	Relational schema representation

S

SGML	Standard generalized markup language
SQL	Structured query language
SUB	Sub-class
SRC	Secondary relationship class
SSC	Secondary sub-class
SST	Secondary strong class

U

UDT	User-defined type
UML	Unified modeling language

W

W3C	World Wide Web consortium

X

XML	extensible markup language
XSD	XML Schema definition language

Bibliography

Abelló, A., Oliva, M., Rodríguez, M. E., and Saltor, F. (1999). The syntax of BLOOM99 schemas. Technical Report LSI-99-34-R, Universitat Politcnica de Catalunya, Dept Llenguatges i Sistemes Informtics, Universitat Politcnica de Catalunya.

Abelló, A. and Rodríguez, E. (2000). Describing BLOOM99 with regard to UML semantics. In *JISBD*, pages 307–320.

Abu-Hamdeh, R., Cordy, J. R., and Martin, P. (1994). Schema translation using structural transformation. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press.

Akoka, J., Comyn-Wattiau, I., and Lammari, N. (1999). Relational database reverse engineering: Elicitation of generalization hierarchies. In *ER '99: Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 173–185, London, UK. Springer-Verlag.

Al-Kamha, R., Embley, D. W., and Liddle, S. W. (2005). Representing generalization/specialization in XML schema. In Desel, J. and Frank, U., editors, *EMISA*, volume 75 of *LNI*, pages 250–263. GI.

Alhajj, R. (1999). Documenting legacy relational databases. In Chen, P. P., Embley, D. W., Kouloumdjian, J., Liddle, S. W., and Roddick, J. F., editors, *ER (Workshops)*, volume 1727 of *Lecture Notes in Computer Science*, pages 161–172. Springer.

Alhajj, R. (2003). Extracting the extended entity-relationship model from a legacy relational database. *Inf. Syst.*, 28(6):597–618.

Alhajj, R. and Polat, F. (2001). Reengineering relational databases to object-oriented: Constructing the class hierarchy and migrating the data. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, pages 335–344, Washington, DC, USA. IEEE Computer Society.

Altova XMLSpy (2008). Altova XMLSpy. `http://altova.com/products/xmlspy/xml_editor.html`.

Ambler, S. (2003). *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA.

Amer-Yahia, S. (1997). From relations to objects: The RelOO prototype. In *Int. Conf. on Data Engineering (ICDE), Industrial session*, Birmingham, England.

Andersson, M. (1994). Extracting an entity relationship schema from a relational database through reverse engineering. In *ER '94: Proceedings of the13th International Conference on the Entity-Relationship Approach*, pages 403–419, London, UK. Springer-Verlag.

Arora, G., Belden, E., Iyer, C., Lee, G., Manikutty, A., Moore, V., Morsi, M., Yeh, H., Yoaz, A., and Yu, Q. (2005). Oracle database application developer's guide - object-relational features, 10g release 2 (10.2), part number b14260-01. Oracle Corporation.

Behm, A. (2001). *Migrating Relational Databases to Object Technology*. PhD thesis, Faculty of Economics, Business Administration and Information Technology, University Of Zurich, Zurich, Switzerland.

Behm, A., Geppert, A., and Dittrich, K. R. (1997). On the migration of relational schemas and data to object-oriented database systems. In Györkös, J., Krisper, M., and Mayr, H. C., editors, *Proc. 5th International Conference on Re-Technologies for Information Systems*, pages 13–33, Klagenfurt, Austria. Oesterreichische Computer Gesellschaft.

Behm, A., Geppert, A., and Dittrich, K. R. (2000). Algebraic database migration to object technology. In *ER*, pages 440–453.

Berglund, A., Boag, S., Chamberlin, D., Fernndez, M. F., Kay, M., Robie, J., and Simon, J. (2007). XML Path language (XPath) 2.0 W3C recommendation 23 january 2007. `http://www.w3.org/TR/xpath20/`.

Bisbal, J., Lawless, D., Wu, B., and Grimson, J. (1999). Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111.

Boag, S., Chamberlin, D., Fernndez, M. F., Florescu, D., Robie, J., and Simon, J. (2009). XQuery 1.0: An XML query language W3C recommendation. `http://www.w3.org/TR/xquery/`.

Bourret, R. (2005). Mapping DTDs to Databases. `http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html/`.

Carey, M., Florescu, D., Ives, Z., Lu, Y., Shanmugasundaram, J., Shekita, E., and Subramanian, S. (2000a). XPERANTO: Publishing object-relational data as XML. In *WebDB (Informal Proceedings)*, pages 105–110.

Carey, M. J., DeWitt, D. J., and Naughton, J. F. (1993). The OO7 benchmark. *SIGMOD Rec.*, 22(2):12–21.

Carey, M. J., DeWitt, D. J., Naughton, J. F., Asgarian, M., Brown, P., Gehrke, J. E., and Shah, D. N. (1997). The BUCKY object-relational benchmark. *SIGMOD Rec.*, 26(2):135–146.

Carey, M. J., Kiernan, J., Shanmugasundaram, J., Shekita, E. J., and Subramanian, S. N. (2000b). XPERANTO: Middleware for publishing object-relational data as XML documents. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 646–648, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Castellanos, M. (1993). A methodology for semantically enriching interoperable databases. In *BNCOD 11: Proceedings of the 11th British National Conference on Databases*, pages 58–75, London, UK. Springer-Verlag.

Castellanos, M., Saltor, F., and García-Solaco, M. (1994). Semantically enriching relational databases into an object oriented semantic model. In *DEXA '94: Proceedings of the 5th International Conference on Database and Expert Systems Applications*, pages 125–134, London, UK. Springer-Verlag.

Cattell, R. G. G. and Barry, D. K., editors (2000). *The object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Cattell, R. G. G. and Skeen, J. (1992). Object operations benchmark. *ACM Trans. Database Syst.*, 17(1):1–31.

Charatan, Q. and Kans, A. (2005). *Java in Two Semesters*. McGraw-Hill Higher Education; 2 edition, Maidenhead, Berkshire, UK.

Chebotko, A., Atay, M., Lu, S., and Fotouhi, F. (2007). XML subtree reconstruction from relational storage of XML documents. *Data Knowl. Eng.*, 62(2):199–218.

Chen, P. P.-S. (1976). The entity-relationship model-toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36.

Chiang, R. H. L. (1995). A knowledge-based system for performing reverse engineering of relational databases. *Decis. Support Syst.*, 13(3-4):295–312.

Chiang, R. H. L., Barron, T. M., and Storey, V. C. (1993). Performance evaluation of reverse engineering relational databases into extended entity-relationship models. In *ER '93: Proceedings of the 12th International Conference on the Entity-Relationship Approach*, pages 352–363, London, UK. Springer-Verlag.

Chiang, R. H. L., Barron, T. M., and Storey, V. C. (1994). Reverse engineering of relational databases: Extraction of an EER model from relational databases. *Data Knowl. Eng.*, 12(2):107–142.

Chiang, R. H. L., Barron, T. M., and Storey, V. C. (1996). A framework for the design and evaluation of reverse engineering methods for relational databases. *Data Knowl. Eng.*, 21(1):57–77.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

Collins, S. R., Navathe, S. B., and Mark, L. (2002). XML schema mappings for heterogeneous database access. *Information & Software Technology*, 44(4):251–257.

Comyn-Wattiau, I. and Akoka, J. (1996). Reverse engineering of relational database physical schemas. In Thalheim, B., editor, *ER '96: Proceedings of the 15th International Conference on Conceptual Modeling*, volume 1157 of *Lecture Notes in Computer Science*, pages 372–391, London, UK. Springer.

Connolly, T. and Begg, C., editors (2002). *Database Systems. (Third Edition.)*. Addison-Wesley, New York.

Conrad, R., Scheffner, D., and Freitag, J. C. (2000). XML conceptual modeling using UML. In Laender, A. H. F., Liddle, S. W., and Storey, V. C., editors, *Conceptual Modeling - ER 2000, 19th International Conference on Conceptual Modeling, Salt Lake City, Utah, USA, October 9-12, 2000, Proceedings*, volume 1920, pages 558–571. Springer.

Darwen, H. and Date, C. J. (1995). The third manifesto. *SIGMOD Record*, 24(1):39–49.

Date, C. J. (2002). *Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Davis, K. H. and Arora, A. K. (1988). Converting a relational database model into an entity-relationship model. In *Proceedings of the Sixth International Conference on Entity-Relationship Approach*, pages 271–285. North-Holland.

Devarakonda, R. S. (2001). Object-relational database systems - the road ahead. *Crossroads*, 7(3):15–18.

Dobbie, G., Wu, X., Ling, T., and Lee, M. (2000). ORA-SS: Object-relationship-attribute model for semistructured data. Technical Report TR21/00, National University of Singapore, Department of Computer Science, National University of Singapore.

DTD (2009). Document type definition (DTD), W3C recommendation. `http://www.w3schools.com/DTD/`.

Du, W., Lee, M.-L., and Ling, T. W. (2001). XML structures for relational data. In *WISE (1)*, pages 151–160.

Duta, A. C., Barker, K., and Alhajj, R. (2004). Conv2XML: Relational schema conversion to XML nested-based schema. In *ICEIS (1)*, pages 210–215.

Eessaar, E. (2006). Whole-part relationships in the object-relational databases. In Bojkovic, Z. S., editor, *Proceedings of the 10th WSEAS Int. Conf. on COMPUTERS, Vouliagmeni, Athens, Greece*, pages 1263–1268.

Eisenberg, A. and Melton, J. (1999). SQL:1999, formerly known as SQL3. *SIGMOD Record*, 28(1):131–138.

Eisenberg, A., Melton, J., Kulkarni, K., Michels, J.-E., and Zemke, F. (2004). SQL:2003 has been published. *SIGMOD Rec.*, 33(1):119–126.

Elmasri, R. and Navathe, S. B. (2006). *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

eXist-db (2009). eXist-db. `http://exist.sourceforge.net/`.

Fahrner, C. and Vossen, G. (1995a). A survey of database design transformations based on the entity-relationship model. *Data Knowl. Eng.*, 15(3):213–250.

Fahrner, C. and Vossen, G. (1995b). Transforming relational database schemas into object-oriented schemas according to ODMG-93. In *DOOD '95: Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases*, pages 429–446, London, UK. Springer-Verlag.

Fallside, D. and Walmsley, P. (2004). XML schema part 0: Primer second edition. W3C proposed edited recommendation 18 march 2004. `http://www.w3.org/TR/xmlschema-0/`.

Fegaras, L. (2008). Lambda-DB. `http://lambda.uta.edu/lambda-DB/manual/`.

Fernandez, M. F., Morishima, A., Suciu, D., and Tan, W. C. (2001). Publishing relational data in XML: the SilkRoute approach. *IEEE Data Eng. Bull.*, 24(2):12–19.

Fernandez, M. F., Tan, W. C., and Suciu, D. (2000). SilkRoute: trading between relations and XML. *Computer Networks*, 33(1-6):723–745.

Fong, J. (1995). Mapping extended entity relationship model to object modeling technique. *SIGMOD Record*, 24(3):18–22.

Fong, J. (1997). Converting relational to object-oriented databases. *SIGMOD Record*, 26(1):53–58.

Fong, J. and Cheung, S. K. (2005). Translating relational schema into XML schema definition with data semantic preservation and XSD graph. *Information & Software Technology*, 47(7):437–462.

Fong, J., Fong, A., Wong, H. K., and Yu, P. (2006). Translating relational schema with constraints into XML schema. *International Journal of Software Engineering and Knowledge Engineering*, 16(2):201–244.

Fong, J., Pang, F., and Bloor, C. (2001). Converting relational database into XML document. In *DEXA '01: Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, pages 61–65, Washington, DC, USA. IEEE Computer Society.

Fong, J., Wong, H. K., and Cheng, Z. (2003). Converting relational database into XML documents with DOM. *Information & Software Technology*, 45(6):335–355.

Fonkam, M. M. and Gray, W. A. (1992). An approach to eliciting the semantics of relational databases. In *Proceedings of the Fourth International Conference on Advanced Information Systems Engineering*, volume 593, pages 463–480. Springer.

Funderburk, J. E., Kiernan, G., Shanmugasundaram, J., Shekita, E. J., and Wei, C. (2002). XTABLES: bridging relational technology and XML. *IBM Systems Journal*, 41(4):616–641.

Garcia-Molina, H., Ullman, J. D., and Widom, J. (2008). *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA.

Getta, J. R. (1993). Translation of extended entity-relationship database model into object-oriented database model. In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, pages 87–100. North-Holland.

Gogolla, M. (1994). *Extended Entity-Relationship Model: Fundamentals and Pragmatics*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Goldman, R., McHugh, J., and Widom, J. (2000). From semistructured data to XML. *Markup Lang.*, 2(2):153–163.

Grant, E. S., Chennamaneni, R., and Reza, H. (2006). Towards analyzing UML class diagram models to object-relational database systems transformations. In Hamza, M. H., editor, *Databases and Applications*, pages 129–134. IASTED/ACTA Press.

Graves, M. and Goldfarb, C. F. (2002). *Designing XML Databases*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Hainaut, J.-L. (1991). Database reverse engineering: Models, techniques and strategies.

Hainaut, J.-L., Hick, J.-M., Henrard, J., Roland, D., and Englebert, V. (1997). Knowledge transfer in database reverse engineering - a supporting case study. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 194, Washington, DC, USA. IEEE Computer Society.

Hainaut, J.-L., Tonneau, C., Joris, M., and Chandelon, M. (1993). Transformation-based database reverse engineering. In *Proceedings of the 12th International Conference on the Entity-Relationship Approach*, pages 364–375, Dallas, Texas. Springer-Verlag.

Hainaut, J.-L., Tonneau, C., Joris, M., and Chandelon, M. (1994). Schema transformation techniques for database reverse engineering. In *ER '93: Proceedings of the 12th International Conference on the Entity-Relationship Approach*, pages 364–375, London, UK. Springer-Verlag.

Hamilton, G., Cattell, R., and Fisher, M. (1997). *JDBC Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Henrard, J., Hick, J.-M., Thiran, P., and Hainaut, J.-L. (2002). Strategies for data reengineering. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 211, Washington, DC, USA. IEEE Computer Society.

Hohenstein, U. (1996). Using semantic enrichment to provide interoperability between relational and odmg databases. In *International Hong Kong Computer Society Database Workshop*, pages 210–232.

Hohenstein, U. (2000). Supporting data migration between relational and object-oriented databases using a federation approach. In *IDEAS*, pages 371–379.

Hohenstein, U. and Körner, C. (1996). A graphical tool for specifying semantic enrichment of relational databases. In *DS-6: Proceedings of the Sixth IFIP TC-2 Working Conference on Data Semantics*, pages 389–420, London, UK. Chapman & Hall, Ltd.

Hohenstein, U. and Plesser, V. (1996). Semantic enrichment: A first step to provide database interoperability. In *Workshop Fderierte Datenbanken*, pages 3–17, Magdeburg.

Hüsemann, F. (1998). Migration of applications and data: An overview of the ReMiS project. In *Föderierte Datenbanken*, pages 133–142.

Jahnke, J. and Zundorf, A. (1998). Using graph grammars for building the VARLET database reverse engineering environment.

Jahnke, J. H., Schäfer, W., and Zündorf, A. (1996). A design environment for migrating relational to object oriented database systems. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 163–170, Washington, DC, USA. IEEE Computer Society.

JDO (2009). Java data objects (JDO), sun microsystems, inc. `http://java.sun.com/products/jdo/`.

Johannesson, P. (1994). A method for transforming relational schemas into conceptual schemas. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 190–201, Washington, DC, USA. IEEE Computer Society.

Johannesson, P. and Kalman, K. (1989). A method for translating relational schemas into conceptual schemas. In Lochovsky, F. H., editor, *Entity-Relationship Approach to Database Design and Querying, Proceedings of the Eight International Conference on Enity-Relationship Approach, Toronto, Canada, 18-20 October, 1989*, pages 271–285. North-Holland.

Kappel, G., Kapsammer, E., and Retschitzegger, W. (2001). XML and relational database systems - a comparison of concepts. In *International Conference on Internet Computing (1)*, pages 199–205.

Keivani, N. (2006). An investigation into the maturity of object-relational database technology, "the promises, the reality". Master thesis, Northumbria University, School of Computing, Engineering and Information Sciences.

Keller, A. M. and Wiederhold, G. (2001). Penguin: Objects for programs, relations for persistence. In *Succeeding with Object Databases*. John Wiley & Sons.

Kim, W. (1991). *Introduction to object-oriented databases*. MIT Press, Cambridge, MA, USA.

Kleiner, C. and Lipeck, U. W. (2001). Automatic generation of XML DTDs from conceptual database schemas. In *GI Jahrestagung (1)*, pages 396–405.

Krishnamurthy, R., Kaushik, R., and Naughton, J. F. (2004). Efficient XML-to-SQL query translation: where to add the intelligence? In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 144–155. VLDB Endowment.

Krumbein, T. and Kudrass, T. (2003). Rule-based generation of XML Schemas from UML class diagrams. In Tolksdorf, R. and Eckstein, R., editors, *Berliner XML Tage*, pages 213–227. XML-Clearinghouse.

Kudrass, T. and Krumbein, T. (2003). Rule-based generation of XML DTDs from UML class diagrams. In Kalinichenko, L. A., Manthey, R., Thalheim, B., and Wloka, U., editors, *ADBIS*, volume 2798 of *Lecture Notes in Computer Science*, pages 339–354. Springer.

Kurt, A. and Atay, M. (2002). An experimental study on query processing efficiency of native-XML and XML-enabled database systems. In *DNIS '02: Proceedings of the Second International Workshop on Databases in Networked Information Systems*, pages 268–284, London, UK. Springer-Verlag.

Laforest, F. and Boumediene, M. (2003). Study of the automatic construction of XML documents models from a relational data model. In *DEXA Workshops*, pages 566–570.

Lammari, N. (1999). An algorithm to extract is-a inheritance hierarchies from a relational database. In *ER '99: Proceedings of the 18th International Conference on Conceptual Modeling*, pages 218–232, London, UK. Springer-Verlag.

Lammari, N., Comyn-Wattiau, I., and Akoka, J. (2007). Extracting generalization hierarchies from relational databases: A reverse engineering approach. *Data Knowl. Eng.*, 63(2):568–589.

Layman, A., Jung, E., Maler, E., Thompson, H. S., Paoli, J., Tigue, J., Mikula, N. H., and Rose, S. D. (1998). XML-data. W3C recommendation, 05 jan 1998. `http://www.w3.org/TR/1998/NOTE-XML-data-0105/`.

Leavitt, N. (2000). Whatever happened to object-oriented databases? *Computer*, 33(8):16–19.

Lee, D. and Chu, W. W. (2000). Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87.

Lee, D., Mani, M., Chiu, F., and Chu, W. W. (2001). Nesting-based relational-to-XML schema translation. In *WebDB*, pages 61–66.

Lee, D., Mani, M., Chiu, F., and Chu, W. W. (2002). NeT and CoT: Translating relational schemas to XML schemas using semantic constraints. In *CIKM*, pages 282–291.

Lee, D., Mani, M., and Chu, W. W. (2003). Solving schema conversion problem between XML and relational models: Semantic approach. Technical report, University of California, University of California, USA.

Lee, H. and Yoo, C. (2000). A form driven object-oriented reverse engineering methodology. *Inf. Syst.*, 25(3):235–259.

Lee, S. H., Kim, S. J., and Kim, W. (2000). The BORD benchmark for object-relational databases. In *DEXA '00: Proceedings of the 11th International Conference on Database and Expert Systems Applications*, pages 6–20, London, UK. Springer-Verlag.

Lewis, J. P. and Neumann, U. (2004). Performance of Java versus C++. Technical report, University of Southern California, Computer Graphics and Immersive Technology Lab, University of Southern California.

Liu, C. and Li, J. (2006). Designing quality XML schemas from E-R diagrams. In Yu, J. X., Kitsuregawa, M., and Leong, H. V., editors, *WAIM*, volume 4016 of *Lecture Notes in Computer Science*, pages 508–519. Springer.

Liu, C., Vincent, M. W., Liu, J., and Guo, M. (2003). A virtual XML database engine for relational databases. In Bellahsene, Z., Chaudhri, A. B., Rahm, E., Rys, M., and Unland, R., editors, *XSym*, volume 2824 of *Lecture Notes in Computer Science*, pages 37–51. Springer.

Lo, A., Alhajj, R., and Barker, K. (2004). Flexible user interface for converting relational data into XML. In Christiansen, H., Hacid, M.-S., Andreasen, T., and Larsen, H. L., editors, *FQAS*, volume 3055 of *Lecture Notes in Computer Science*, pages 418–431. Springer.

Maatuk, A., Ali, M. A., and Rossiter, B. N. (2008a). An integrated approach to relational database migration. In *IC-ICT '08: Proceedings of International Conference on Information and Communication Technologies*, page 6pp, Pakistan. PAK. In Press.

Maatuk, A., Ali, M. A., and Rossiter, B. N. (2008b). Relational database migration: A perspective. In Bhowmick, S. S., Küng, J., and Wagner, R., editors, *DEXA*, volume 5181 of *Lecture Notes in Computer Science*, pages 676–683. Springer.

Maatuk, A., Ali, M. A., and Rossiter, B. N. (2008c). Semantic enrichment: The first phase of relational database migration. In *International Conference on Systems, Computing Sciences and Software Engineering (SCS2 08)*, page 6pp, USA. Springer. In Press.

Maatuk, A., Ali, M. A., and Rossiter, B. N. (2010). Converting relational databases into object-relational databases. *Journal of Object Technology*, page 17pp. In Press.

Malki, M., Flory, A., and Rahmouni, M. K. (2001). Static and dynamic reverse engineering of relational database applications: A form-driven methodology. In *AICCSA*, pages 191–196. IEEE Computer Society.

Malki, M., Flory, A., and Rahmouni, M. K. (2002). Extraction of object-oriented schemas from existing relational databases: a form-driven approach. *Informatica, Lith. Acad. Sci.*, 13(1):47–72.

Marcos, E., Vela, B., and Cavero, J. M. (2003). A methodological approach for object-relational database design using UML. *Software and System Modeling*, 2(1):59–75.

Marcos, E., Vela, B., Cavero, J. M., and Caceres, P. (2001). Advances in databases and information systems, 5th east european conference, adbis 2001, vilnius, lithuania, september 25-28. In Caplinskas, A. and Eder, J., editors, *ADBIS*, volume 1 of *Lecture Notes in Computer Science*, pages 195–209. Springer.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. (1997). Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66.

Missaoui, R., Gagnon, J.-M., and Godin, R. (1995). Mapping an extended entity-relationship schema into a schema of complex objects. In *Object-Oriented and Entity-Relationship Modelling*, pages 204–215.

Missaoui, R., Godin, R., and Sahraoui, H. (1998). Migrating to an object-oriented databased using semantic clustering and transformation rules. *Data & Knowledge Engineering*, 27(1):97–113.

Mok, W. and Paper, D. (2001). On transformations from UML models to object-relational databases. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, volume 3, page 3046, Washington, DC, USA. IEEE Computer Society.

Mok, W. Y. (2007). Designing nesting structures of user-defined types in object-relational databases. *Inf. Softw. Technol.*, 49(9-10):1017–1029.

Monk, S., Mariani, J. A., Elgalal, B., and Campbell, H. (1996). Migration from relational to object-oriented databases. *Information & Software Technology*, 38(7):467–475.

Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., and Wong, K. (2000). Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA. ACM.

Narasimhan, B., Navathe, S. B., and Jayaraman, S. (1993). On mapping ER models into OO schemas. In *ER*, pages 402–413.

Navathe, S. B. and Awong, A. M. (1988). Abstracting relational and hierarchical data with a semantic data model. In March, S. T., editor, *Proceedings of the Sixth International Conference on Entity-Relationship Approach*, pages 305–333, Amsterdam, Netherlands. North-Holland Publishing Co.

Objectivity (2009). Objectivity home page. Objectivity. `http://objectivity.com/Products/Products.shtml`.

ObjectStore (2009). Objectstore home page. ObjectStore. `http://objectstore.com/datasheet/index.ssp`.

OEM (2009). Object Exchange Model (OEM). `http://infolab.stanford.edu/ mchughj/oemsyntax/oemsyntax.html`.

OMG (2009). Unified Modeling Language (UML), version 2.0. `http://www.uml.org/`.

Orenstein, J. A. and Kamber, D. N. (1995). Accessing a relational database through an object-oriented database interface. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 702–705, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Pardede, E., Rahayu, J. W., and Taniar, D. (2003). New SQL standard for object-relational database applications. In *SIIT*, pages 191–203, Delft, The Netherlands. IEEE.

Pardede, E., Rahayu, J. W., and Taniar, D. (2004). Mapping methods and query for aggregation and association in object-relational database using collection. In *ITCC (1)*, volume 1, pages 539–, Las Vegas, Nevada, USA. IEEE Computer Society.

Parent, C. and Spaccapietra, S. (2000). Database integration: The key to data interoperability. In Papazoglou, M. P., Spaccapietra, S., and Tari, Z., editors, *Advances in Object-Oriented Data Modeling*, pages 221–253. MIT Press.

Pérez, J., Anaya, V., Cubel, J. M., Ramos, I., and Carsí, J. A. (2003). Data reverse engineering of legacy databases to object oriented conceptual schemas. *Electronic Notes in Theoretical Computer Science*, 72(4).

Petit, J.-M., Kouloumdjian, J., Boulicaut, J.-F., and Toumani, F. (1994). Using queries to improve database reverse engineering. In *ER '94: Proceedings of the13th International Conference on the Entity-Relationship Approach*, pages 369–386, London, UK. Springer-Verlag.

Petit, J.-M., Toumani, F., Boulicaut, J., and Kouloumdjian, J. (1996). Towards the reverse engineering of denormalized relational databases. In Society, I. C., editor, *Proc. 12th International Conference on Data Engineering*, pages 218–227, New Orleans. S. Su.

Pigozzo, P. and Quintarelli, E. (2005). An algorithm for generating XML schemas from ER schemas. In *SEBD*, pages 192–199.

Premerlani, W. J. and Blaha, M. R. (1994). An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49.

Ramanathan, S. and Hodges, J. (1996). Reverse engineering relational schemas to object-oriented schemas. Technical Report 960701, Department of Computer Science, Mississippi State University.

Ramanathan, S. and Hodges, J. E. (1997). Extraction of object-oriented structures from existing relational databases. *SIGMOD Record*, 26(1):59–64.

Roguewave (2006). Rogue wave software. Rogue Wave. `http://www.roguewave.com`.

Routledge, N., Bird, L., and Goodchild, A. (2002). UML and XML schema. *Australian Computer Science Communications*, 24(2):157–166.

Rumbaugh, J. R., Blaha, M. R., Lorensen, W., Eddy, F., and Premerlani, W. (1990). *Object-Oriented Modeling and Design*. Prentice-Hall.

Runapongsa, K., Patel, J. M., Jagadish, H. V., Chen, Y., and Al-Khalifa, S. (2006). The michigan benchmark: towards XML query performance diagnostics. *Inf. Syst.*, 31(2):73–97.

Saltor, F., Castellanos, M., and Garcia-Solaco, M. (1991). Suitability of datamodels as canonical models for federated databases. *SIGMOD Rec.*, 20(4):44–48.

Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J., and Busse, R. (2001). The XML benchmark project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands.

Shanmugasundaram, J., Shekita, E., Barr, R., Carey, M., Lindsay, B., Pirahesh, H., and Reinwald, B. (2001). Efficiently publishing relational data as XML documents. *The VLDB Journal*, 10(2-3):133–154.

Singh, A., Kahlon, K. S., Singh, J., Singh, R., Sharma, S., and Kaur, D. (2004). Mapping relational database schema to object-oriented database schema. In *International Conference on Computational Intelligence*, pages 153–155.

Sousa, P., de Jesus, L. P., Pereira, G., and e Abreu, F. B. (2002). Clustering relations into abstract er schemas for database reverse engineering. *Science of Computer Programming*, 45(2-3):137–153.

Soutou, C. (1996). Extracting N-ary relationships through database reverse engineering. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 392–405.

Soutou, C. (1998a). Inference of aggregate relationships through database reverse engineering. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 135–149, London, UK. Springer-Verlag.

Soutou, C. (1998b). Relational database reverse engineering: Algorithms to extract cardinality constraints. *Data Knowl. Eng.*, 28(2):161–207.

Soutou, C. (2001). Modeling relationships in object-relational databases. *Data Knowl. Eng.*, 36(1):79–107.

Stonebraker, J. M., Brown, P., and Moore, D. (1999). *Object-Relational DBMSs: The Next Great Wave and Object-Relational DBMSs: Tracking the Next Great Wave.* Morgan Publishers.

Takahashi, T. and Keller, A. M. (1993). Querying heterogeneous object views of a relational database. In *Int. Symp. on Next Generation Database Systems and Their Applications (NDA)*, pages 34–41, Fukuoka, Japan.

Takahashi, T. and Keller, A. M. (1994). Implementation of object view query on a relational database. In *Data and Knowledge Systems for Manufacturing and Engineering*, Hong Kong.

Tamino XML Server (2009). Tamino XML Server. `http://www.softwareag.com/corporate/products/wm/tamino/default.asp`.

Tari, Z., Bukhres, O. A., Stokes, J., and Hammoudi, S. (1997). The reengineering of relational databases based on key and data correlations. In *DS-7*, pages 184–.

Tari, Z. and Stokes, J. (1997). Designing the reengineering services for the DOK federated database system. In Gray, W. A. and Larson, P.-Å., editors, *ICDE*, pages 465–475. IEEE Computer Society.

Teorey, T. J., Wei, G., Bolton, D. L., and Koenig, J. A. (1989). ER model clustering as an aid for user communication and documentation in database design. *Commun. ACM*, 32(8):975–987.

TopLink (2006). Oracle toplink. `http://www.oracle.com/technology/products/ias/toplink/index.html`.

Turau, V. (1999). Making legacy data accessible for XML applications. Web page.

Urban, S. D. and Dietrich, S. W. (2003). Using UML class diagrams for a comparative analysis of relational, object-oriented, and object-relational database mappings. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 21–25, New York, NY, USA. ACM Press.

Urban, S. D., Dietrich, S. W., and Tapia, P. (2001). *Succeeding with Object Databases: A Practical Look at Today's Implementations with Java and XML*, chapter Mapping UML Diagrams to Object-Relational Schemas in Oracle 8, pages 29–51. John Wiley and Sons, Ltd.

Urban, S. D., Tjahjadi, M., and Shah, J. J. (2000). A case study in mapping conceptual designs to object-relational schemas. *Concurrency - Practice and Experience*, 12(9):863–907.

Valentine, C., Tittel, E., and Dykes, L. (2002). *XML Schemas*. SYBEX Inc., Alameda, CA, USA.

Valikov, A., Kazakos, W., and Schmidt, A. (2001). Building updateable XML views on top of relational databases. In Lasker, G. E. and Dahanayake, A., editors, *Proceedings of the International Symposium on Systems Integration (INTERSYMP 2001)*, pages VII–1 – VII–8. The International Institute for Advanced Studies in Systems Research and Cybernetics.

Vela, B. and Marcos, E. (2003). Extending UML to represent XML schemas. In Eder, J. and Welzer, T., editors, *CAiSE Short Paper Proceedings*, volume 74 of *CEUR Workshop Proceedings*. CEUR-WS.org.

W3C (2008). World Wide Web Consortium (W3C). XML Schema. `http://www.w3.org/XML/Schema`.

W3C (2009). World Wide Web Consortium (W3C). `http://www.w3.org`.

Wang, C. (2004). COCALEREX: An engine for converting catalog-based and legacy relational databases into XML. Master thesis, The University of Calgary, Department of Computer Science, Calgary, Alberta.

Wang, C., Lo, A., Alhajj, R., and Barker, K. (2004). Converting legacy relational database into XML database through reverse engineering. In *ICEIS (1)*, pages 216–221.

Wang, C., Lo, A., Alhajj, R., and Barker, K. (2005). Novel approach for reengineering relational databases into XML. In *ICDE Workshops*, page 1284.

Wiener, J. L. (1996). *Algorithms for loading object databases*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA.

Wiener, J. L. and Naughton, J. F. (1994). Bulk loading into an OODB: A performance study. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 120–131, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

XML (2008). World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML`.

Yan, L.-L. and Ling, T. W. (1993). Translating relational schema with constraints into OODB schema. In *Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, pages 69–85. North-Holland.

Yoshikawa, M., Amagasa, T., Shimura, T., and Uemura, S. (2001). XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Techn.*, 1(1):110–141.

Zhang, X. and Fong, J. (2000). Translating update operations from relational to object-oriented databases. *Information & Software Technology*, 42(3):197–210.

Zhang, X., Zhang, Y., Fong, J., and Jia, X. (1999). Transforming RDB schema into well-structured OODB schema. *Information & Software Technology*, 41(5):275–281.